Synthesis and Axiomatisation for Structural Equivalences in the Petri Box Calculus

Martin Hesketh

24th June 1998

Acknowledgements

The author would like to thank his supervisor, Dr. Maciej Koutny, for all the support and encouragement that helped to ensure the completion of this thesis.

The author would also like to thank his manager at Nortel, Jonathan Atkinson, for his flexibility and understanding when allowing extra time to work on the thesis as deadlines were looming.

Abstract

The Petri Box Calculus (PBC) consists of an algebra of box expressions, and a corresponding algebra of boxes (a class of labelled Petri nets). A compositional semantics provides a translation from box expressions to boxes. The synthesis problem is to provide an algorithmic translation from boxes to box expressions. The axiomatisation problem is to provide a sound and complete axiomatisation for the fragment of the calculus under consideration, which captures a particular notion of equivalence for boxes.

There are several alternative ways of defining an equivalence notion for boxes, the strongest one being net isomorphism. In this thesis, the synthesis and axiomatisation problems are investigated for net semantic isomorphism, and a slightly weaker notion of equivalence, called *duplication equivalence*, which can still be argued to capture a very close structural similarity of concurrent systems the boxes are supposed to represent.

In this thesis, a structured approach to developing a synthesis algorithm is proposed, and it is shown how this may be used to provide a framework for the production of a sound and complete axiomatisation. This method is used for several different fragments of the Petri Box Calculus, and for generating axiomatisations for both isomorphism and duplication equivalence. In addition, the algorithmic problems of checking equivalence of boxes and box expressions, and generating proofs of equivalence are considered as extensions to the synthesis algorithm.

Keywords: Petri nets, process algebra, equivalence, axiomatisation, synthesis, structure, isomorphism

Contents

Chapter 1

Introduction

This chapter introduces the Petri Box Calculus, and the synthesis and axiomatisation problems that are investigated in the remainder of the thesis. Section 1.2 presents the algebra of box expressions, together with an informal description of the intended semantics of each type of expression. Labelled nets, which are used to give a formal semantics to box expressions are introduced in Section 1.3.2. The translation from expressions to nets is described in Section 1.3, and some notions of equivalence in the Petri Box Calculus, based on the structure of nets, are given in Section 1.4. The synthesis and axiomatisation problems are introduced, followed by a survey of some related work on the derivation of algebraic representations for net based models and axiomatisations for process algebra based models. Finally, a summary of the remainder of the thesis is given in Section 1.7.

1.1 The Petri Box Calculus

Formal models for concurrent systems are used for specifying, modelling, designing, simulating and verifying complex systems that involve multiple threads of control and communication between concurrent components of the system. A specification for a concurrent system consists of set of properties, expressed in a formal manner, that should hold for any implementation of the

system. Properties may include the absence of deadlocks, termination (or non-termination), and the ability to perform certain actions in particular states. Formal models for concurrent systems can be used to model existing systems, or to design new systems. Verification tools can be used to check properties, such as those used in specifications, of a formal model. Simulation can be used to examine the behaviour of a formal model, and is useful for testing and debugging the design of a concurrent system. Examples of areas in which formal models of concurrency have been applied include the verification of communications protocols, the giving of a formal semantics to concurrent programming languages, the modelling of workflow in businesses, and the simulation of the interaction between a pilot and his aircraft.

Two important models for concurrency are Petri nets [46] and process algebras [2, 41]. The Petri net model is graphical in nature, whereas process algebras use an algebraic approach. Petri nets have a partial order, or "true concurrency" behaviour, allowing reasoning about causal relationships between events. This allows systems to be debugged easily – for example, in a Petri net model of a system that could deadlock, it is possible to find the chain of events that lead to the deadlocking behaviour. In comparison, process algebraic models are generally based on less rich interleaving behaviours, where causality information is not available. During the simulation of Petri nets, the current state of the system, and the available set of actions that can be performed are easily visible, due to the graphical form of the Petri net model. However, it may be more difficult to follow the simulation where the system is so large that it is not practical to use a global view of the entire net.

Both Petri net and process algebra models for concurrency have well developed tools for the automatic verification of properties. Petri net tools are generally based on the generation of structures that contain information about the reachability of the various states of the system. Process algebra verification tools are usually based around an axiomatisation of the algebra, together with procedures for applying the axioms. Process algebras consist of a set of

operators, where each operator corresponds to a particular type of behaviour. In comparison, Petri nets allow the arbitrary interconnection of components, which in some respects, gives much greater flexibility.

The major deficiency of the Petri net model is that it does not readily support the composition of nets. This makes it more difficult to produce modular designs for systems than in a process algebra based framework. Without compositionality, it is not possible to take a top-down decompositional approach to designing a system, or to design systems at different levels of abstraction, where a high level, less detailed design can be refined to a lower level more detailed design. Of course, with careful planning, it is possible to model systems in a modular fashion using Petri nets. In doing so, some of the flexibility of the expressiveness of nets is inevitably lost.

The Petri Box Calculus [5, 6], one of the results of the Esprit Basic Research Action, DEMON and its successor, CALIBAN, has been designed to provide the advantages of both Petri nets and process algebras. The calculus consists of the box algebra, a process algebraic domain of box expressions, and a semantic domain of Petri boxes, classes of labelled Petri nets. A compositional semantics provides a translation from box expressions to Petri boxes. Earlier approaches to giving a Petri net interpretation to a process algebra, [24, 48], have been based on algebras with an existing semantics in a model other than Petri nets. Note that the design of the Petri Box Calculus does not preclude a semantics being given in purely algebraic terms [35, 37]. For example, [35] gives a partial order operational semantics for box expressions, which is consistent with the corresponding partial order semantics of Petri boxes.

One of the aims of the box calculus is to allow the semantics of high level programming constructs to be simulated, verified and reasoned about at the level of Petri nets. In this respect, the box algebra lies midway between high level concurrent programming languages such as `occam` [31], and B(PN)^2, [7]. Semantics for both of these high level languages have been given using the Box Calculus [7, 30].

The class of nets that belong to the domain of Petri boxes is very much smaller than the general class of labelled Petri nets. The existence of translations from high level languages such as occam to the Petri Box Calculus demonstrate that the calculus is sufficiently expressive for real applications. An analogy can be drawn with standard sequential programming languages, where Petri nets can be seen as an assembly language, the Petri Box Calculus an intermediate p-code, and languages such as B(PN)^2 and occam as high level languages. High level languages are compiled into an intermediate p-code, then into assembly language. There is usually a simple translation from programs represented by p-code into assembly language. Developing systems at the lowest level (assembly language/Petri nets) will often give more flexibility, and more compact and efficient designs. However, the disadvantages are the difficulty of maintaining, debugging and modifying the system.

1.2 Syntax

Table 1.1 gives the BNF description of the algebra of box expressions. This algebra forms the syntactic domain of the Petri Box Calculus, which is described in detail in [5]. In this section, an informal description of the intended semantics of each of the operators in Table 1.1 is given. Section 1.3 introduces a formal semantics for box expressions based on a translation from expressions to labelled Petri nets. [35] gives a detailed and formal description of an operational semantics for box expressions, based on annotated expressions.

In the following description of the operators in Table 1.1, the notions of executing an expression, and the successful or unsuccessful termination of an expression are discussed. In effect, expressions can be considered to be concurrent programs whose execution proceeds by performing basic actions which are regarded as atomic. Since there is an element of concurrency, it is possible that sets of actions are executed simultaneously. The execution of an expression terminates when no more actions can be performed. For

$E ::=$	α	Atomic action
	$E \parallel E$	Parallel composition
	$E \,\square\, E$	Choice composition
	$E; E$	Sequential composition
	$[E * E * E]$	Iteration
	E **rs** a	Restriction
	stop	Stop
	E **sy** a	Synchronisation
	$[a : E]$	Scoping
	$E[f]$	Relabelling
	X	Variable
	$E[X \leftarrow E]$	Refinement
	$\mu X.E$	Recursion

Table 1.1: Box expression syntax

certain expressions, the execution may terminate unsuccessfully by entering a deadlocked state.

An infinite set of basic action names, $\mathcal{B}$ is assumed. For the purposes of the investigation into the synthesis and axiomatisation problems, the lower case letters (*i.e.* a,b,c,...) will generally be used. However, it will be usual in practical applications of the Petri Box Calculus to use more meaningful names. The $\widehat{\ }$ symbol is used to denote conjugation, which is a bijection, $\widehat{\ }: \mathcal{B} \rightarrow \mathcal{B}$ such that for any basic action b, $\widehat{\widehat{b}} = b$, and $\widehat{b} \neq b$. For example, the basic actions b and $\widehat{b}$ are conjugates of each other. The mapping $\widehat{\ }$ can be extended to sets and multisets[1] of actions, A, with $\widehat{A} = \{\widehat{a} \mid a \in A\}$. It will be seen below that conjugate actions are required for the synchronisation and scoping operations.

[1] multisets are an extension of sets to allow multiple occurrences of elements. Section 1.3.1 gives a formal definition.

Atomic action ($E ::= \alpha$)

An atomic action, α is a finite multiset of basic actions. For example:

$$\alpha = \{a, b, \widehat{a}, a, c\}$$

When an atomic action expression is executed, every basic action in the multiset is performed simultaneously, and the execution of the expression successfully terminates.

If an atomic action consists of a single basic action, then, as shorthand notation, the braces will usually be omitted – *i.e.* $\alpha = a$ is shorthand for $\alpha = \{a\}$. Unless illustrating some point requiring the use of multiactions, future examples will contain only atomic actions consisting of a single basic action.

Parallel composition ($E ::= E \parallel E$)

When an expression of the form, $E_1 \parallel E_2$ is executed, the subexpressions E_1 and E_2 are executed concurrently. Any concurrent execution must arise as the result of a parallel composition operator. The execution of $E_1 \parallel E_2$ terminates when the execution of both E_1 and E_2 has terminated. If either E_1 or E_2 deadlocks during execution, then $E_1 \parallel E_2$ will eventually deadlock, resulting in an unsuccessful termination.

Choice composition ($E ::= E \,\square\, E$)

When an expression of the form, $E_1 \,\square\, E_2$ is executed, either subexpression E_1 or subexpression E_2 is executed. Once execution of one subexpression has begun, no part of the other subexpression can be executed. However, the choice of subexpression is not fixed – for example, if a choice expression $E_1 \,\square\, E_2$ appears as a subexpression of an iteration expression then it is possible for E_1 to be chosen during the first iteration and E_2 to be chosen during the next iteration. The execution of a choice composition expression terminates when

the execution of the chosen subexpression terminates, and the success of the execution depends on the success of the execution of that subexpression.

Sequential composition ($E ::= E; E$)

An expression of the form, $E_1; E_2$ is executed by executing subexpression E_1 followed by subexpression E_2. The execution of E_2 cannot begin until the execution of E_1 has terminated, and will never begin if the execution of E_1 terminates unsuccessfully. The execution of $E_1; E_2$ terminates successfully if and only if the execution of both E_1 and E_2 is successful.

Iteration ($E ::= [E * E * E]$)

When an expression of the form $[E_1 * E_2 * E_3]$ is executed; subexpression E_1 is executed once, then E_2 is executed zero or more times (*i.e.* it is possible that E_2 is not executed at all); finally, subexpression E_3 is executed once.

The form of the iteration expression is partly motivated by the Petri net semantics given to box expressions. The inclusion of subexpressions E_1 and E_3 in $[E_1 * E_2 * E_3]$ ensure that some of the desirable properties for Petri boxes also hold for nets derived from iteration expressions. An iteration operator was not included in the initial presentation of the Petri Box Calculus, [6], where it was left to the recursion operator to provide the capability of infinite behaviour. The iteration operator first appeared in [5], where the semantics allowed unsafe nets to be obtained. The net semantics of iteration were updated in [4] to guarantee that every net derived from an iteration expression was safe.

Restriction ($E ::= E$ **rs** a)

Restriction on a basic action name, a, prevents the execution of all atomic actions within the scope of the restriction operator that contain a basic action a, or its conjugate, $\widehat{a}$. For example, in:

$$((\{a, b\} \parallel c) \textbf{ rs } a) \,\square\, \{a, d\}$$

the execution of the atomic action $\{a, b\}$ is prevented by the restriction operator. However, the atomic action $\{a, d\}$ may still execute as it is not in the scope of the restriction operation. The restriction operator is most often used in conjunction with the synchronisation operator to provide scoping.

Synchronisation ($E ::= E$ **sy** a)

Consider an expression E synchronised by a basic action a. Whenever E has the capability to execute an a and $\widehat{a}$ concurrently, then E **sy** a also offers the alternative of synchronising the execution of the a and $\widehat{a}$ actions. When a pair of conjugate basic actions are executed synchronously, the execution of the pair of basic actions that synchronise is hidden, but the system ends up in the same state as if that pair of actions were executed normally. In this sense, synchronisation can be seen as a generalisation of the choice composition operator, allowing a choice between normal and hidden execution of synchronised actions. While synchronisation occurs between pairs of basic actions, it is pairs of atomic actions that are executed synchronously. For example, the expression:

$$(\{a, b\} \parallel \{\widehat{a}, c, d\})\ \mathbf{sy}\ a$$

could either execute the two atomic actions $\{a, b\}$ and $\{\widehat{a}, c, d\}$ concurrently, or execute the multiset of basic actions, $\{b, c, d\}$ in a single step, where the basic action b originates from $\{a, b\}$ and the c and d basic actions come from the action $\{\widehat{a}, c, d\}$. In addition to the synchronisation of pairs of actions, the synchronisation operator permits multi-way synchronisation where multiple atomic actions are executed synchronously. This is a consequence of the multiset representation for atomic actions. For example, in the expression:

$$(\{a, b, b\} \parallel \{\widehat{b}, c\} \parallel \{d, \widehat{b}\})\ \mathbf{sy}\ b$$

all three atomic actions are synchronised, permitting the execution of the expression to complete in a single step by performing the multiset of basic actions $\{a, c, d\}$. The semantics for the synchronisation operator are given

in Section 1.3. An alternative, more intuitive, but equivalent semantics for synchronisation is described in Chapter 4 together with an investigation into some properties of synchronisation.

Scoping ($E ::= [a : E]$)

The semantics of the scoping operator can be defined syntactically in terms of the synchronisation and restriction operators:

$$[a : E] = E \textbf{ sy } a \textbf{ rs } a$$

Whereas synchronisation provides the choice between executing synchronised actions normally or synchronously, the scoping operator forces the synchronous execution. It is possible to obtain deadlocking behaviour using the scoping operator. A deadlock occurs when a pair of actions that cannot be executed concurrently are scoped. For example, the expression $[a : b; a; \widehat{a}]$ deadlocks after performing the atomic action b because the a and $\widehat{a}$ cannot be executed concurrently.

Although the scoping operator does not increase the expressive power of the syntax in Table 1.1, the inclusion of scoping is nevertheless important, as this operator represents the normal use of the synchronisation and restriction operators. In this respect, the scoping operator can be seen as a shorthand notation.

Stop ($E ::= \textbf{stop}$)

The semantics of **stop** can be defined syntactically in terms of either the scoping or restriction operators:

$$\textbf{stop} = [a : a] = a \textbf{ rs } a$$

Hence, **stop** does not make any contribution to the expressive power of the syntax in Table 1.1. The main use of **stop** is to enforce an explicit deadlock, or unsuccessful termination.

Relabelling ($E ::= E[f]$)

The relabelling operator provides a means of associating a relabelling function with a box expression. The relabelling function acts on basic action names and variables. By changing this function, a class of relabelled expressions can be obtained. There is some interplay between relabelling and operators such as synchronisation and restriction. For example, in $(\widehat{a} \parallel b)$ **sy** $a[b \rightarrow a]$ no synchronisation takes place. The relabelling operator is most useful when used in conjunction with recursion. It is possible to show that for any expression that involves relabelling, but not recursion, there is an equivalent expression that contains neither relabelling nor recursion operators. Hence, relabelling does not increase the expressive power of the calculus, unless used in conjunction with the recursion operator.

Variable ($E ::= X$)

Variables are used in conjunction with the refinement or recursion operators. When every variable in an expression occurs in the scope of an enclosing refinement or recursion operator acting on that variable, the expression is said to be closed. The compositional semantics given in Section 1.3 only defines the behaviour of closed expressions.

Refinement ($E ::= E[X \leftarrow E]$)

The purpose of refinement is to provide a basis for the recursion operator. The idea of refinement is that in an expression such as $E_1[X \leftarrow E_2]$, the behaviour of open occurrences of the variable X in E_1 is obtained by executing E_2 in their place. Refinement is not exactly the same as syntactic substitution because there is an interplay between refinement and operators such as synchronisation, restriction and scoping. For example, if refinement was equivalent to syntactic substitution then in the expression:

$$((a \parallel X \parallel b) \textbf{ sy } a)[X \leftarrow \widehat{a}]$$

there would be the possibility of a synchronisation between the a and $\hat{a}$ actions. However, the intended semantics of refinement do not permit such a synchronisation to occur.

Recursion ($E ::= \mu X.E$)

Recursion is defined inductively in terms of a succession of refinements. For example, for $\mu X.E$, the inductive definition:

$$\begin{aligned} E_0 &= \textbf{stop} \\ E_{i+1} &= E[X \leftarrow E_i] \end{aligned}$$

gives a sequence of expressions $E_0, E_1, E_2, ...$ whose behaviour successively approximates that of $\mu X.E$. In this respect it is very difficult to explicitly describe how the execution of expressions involving the recursion operator will proceed. What is certainly clear is that the inclusion of recursion in the box expression syntax in Table 1.1 greatly enhances the expressive power of the calculus. In general, expressions involving the recursion operator result in infinite nets. For this reason, recursion is not investigated further in this thesis. For more details on recursion in the Petri Box Calculus, the reader is referred to [4] and [12].

1.3 Semantics

In this section, a formal basis for describing the semantics of the syntax in Table 1.1 is presented. The semantics are given in terms of labelled Petri nets. Some examples of the translation from expressions to nets are presented, and the general form of the translation is discussed. However, detailed presentations of the semantics are given only for a subset of the syntax in Table 1.1. A complete description of the semantics is contained in [5].

A formal description is given for multisets, which among other things are used to represent atomic actions. A definition of labelled Petri nets is given

in Section 1.3.2, followed by a brief description of the execution behaviour of such nets. Some operators, used to compose labelled nets are described in Section 1.3.4. The semantics of the syntax in Table 1.1 are given in terms of these operators. Finally, in Section 1.3.5, the ideas behind the translation from expressions to nets are discussed, and the semantics for some of the operators in Table 1.1 are presented.

1.3.1 Multisets

A particular set S can be described using a characteristic function[2], $\mathcal{I} : U \rightarrow \{0, 1\}$. The domain of $\mathcal{I}$, U is known as the universe of S, and is the set of all elements that could conceivably be present in S. For all $u \in U$, the value of $\mathcal{I}(u)$ indicates whether u is actually present in S, with $\mathcal{I}(u) = 1$ if and only if $u \in S$:

$$\mathcal{I}(u) = \begin{cases} 1 & \text{if } u \in S \\ 0 & \text{otherwise} \end{cases}$$

For example, suppose $S = \{1, 3, 4\}$ is a set of natural numbers (hence the universe of S is the set of all natural numbers, $\mathcal{N}$), and $\mathcal{I} : \mathcal{N} \rightarrow \{0, 1\}$ is the characteristic function for S, then $\mathcal{I}(1) = \mathcal{I}(3) = \mathcal{I}(4) = 1$ and $\mathcal{I}(0) = \mathcal{I}(2) = \mathcal{I}(n) = 0$ for all $n \geq 5$.

The functional notation for sets is more formal than the standard set notation as it makes the universe of the set explicit. A definition of multisets is obtained by extending the range of the characteristic function from $\{0, 1\}$ to the set of natural numbers, $\mathcal{N}$. Intuitively, this allows each element to appear multiple times in the set. Hence, a multiset with universe X is a function, $\mu : X \rightarrow \mathcal{N}$, where for each $x \in X$, $\mu(x)$ gives the multiplicity of the element x. Multisets will usually be written in standard set notation – for example a multiset $\mu : \{a, b, c, d\} \rightarrow \mathcal{N}$ with $\mu(a) = 2$, $\mu(b) = 1$, $\mu(c) = 0$, and $\mu(d) = 3$ can be written as $\{a, a, b, d, d, d\}$.

[2]This notation is most often encountered in proofs that the number of subsets of a set S is $2^{|S|}$.

Let μ_1 and μ_2 be multisets with universe X. The standard set operations union ($\cup$), intersection ($\cap$) and difference ($-$), together with multiset sum ($+$) and multiplication ($\cdot$) operations can be defined for μ_1 and μ_2. For all $x \in X$ and $n \in \mathcal{N}$:

$$\begin{aligned}
(\mu_1 \cup \mu_2)(x) &= \max(\mu_1(x), \mu_2(x)) \\
(\mu_1 \cap \mu_2)(x) &= \min(\mu_1(x), \mu_2(x)) \\
(\mu_1 - \mu_2)(x) &= \begin{cases} \mu_1(x) - \mu_2(x) & \text{if} \mu_1(x) \geq \mu_2(x) \\ 0 & \text{otherwise} \end{cases} \\
(\mu_1 + \mu_2)(x) &= \mu_1(x) + \mu_2(x) \\
(n \cdot \mu)(x) &= n \cdot (\mu(x))
\end{aligned}$$

Let μ be a multiset with universe X, and $X' \subseteq X$ be a subset of X. The multiset $\mu \mid X'$ denotes μ restricted to the domain (or universe) X'. This follows the usual notation for restricting a function to a particular domain – *i.e.* where $f \mid_A$ denotes the function f restricted to the domain A. For example, let $X = \{a, b, c, d\}$, $\mu_1 = \{a, a, b, c, d, d, d\}$ and $\mu_2 = \{a, b, b, d, d\}$ then:

$$\begin{aligned}
\mu_1 \cup \mu_2 &= \{a, a, b, b, c, d, d, d\} & \qquad \mu_1 \cap \mu_2 &= \{a, b, d, d\} \\
\mu_1 - \mu_2 &= \{a, c, d\} & \qquad \mu_2 - \mu_1 &= \{b\} \\
\mu_1 + \mu_2 &= \{a, a, a, b, b, b, c, d, d, d, d, d\} & \qquad \mu_1 \mid_{\{a,b\}} &= \{a, a, b\} \\
2 \cdot \mu_2 &= \{a, a, b, b, b, b, d, d, d, d\}
\end{aligned}$$

It follows directly from the definition of multisets that every set can be treated as a multiset. It is easy to check that the definition of union, intersection, difference and restriction for multisets is consistent with the definition for sets.

1.3.2 Labelled nets

A labelled net is a tuple, $\Sigma = (S, T, W, \lambda)$, where S and T are sets of places, and transitions respectively, collectively known as nodes. The set of arcs of

the net is given by $W : (S \cup T) \times (S \cup T) \to \mathcal{N}$. $W(n_1, n_2)$ returns a non-zero value n to indicate the presence of an arc from node n_1 to node n_2 with weight n. If $W(n_1, n_2)$ returns 0, then there is no arc from n_1 to n_2.

As a shorthand notation, the set of arcs of the net may be written as a multiset of pairs of nodes, such that (n_1, n_2) appears exactly n times if and only if $W(n_1, n_2) = n$. It is assumed that labelled nets are bipartite, with bipartition (S, T). Therefore there are no arcs between pairs of places, or pairs of transitions:

$$\forall s_1, s_2 \in S, t_1, t_2 \in T : W(s_1, s_2) + W(t_1, t_2) = 0$$

λ is a labelling function, such that a place is labelled e for an entry place, $\emptyset$ for an internal place, and x for an exit place. ${}^\bullet\Sigma$ and $\Sigma^\bullet$ are the set of entry and exit places respectively:

$$\begin{aligned} {}^\bullet\Sigma &= \{s \in S \mid \lambda(s) = \mathsf{e}\} \\ \Sigma^\bullet &= \{s \in S \mid \lambda(s) = \mathsf{x}\} \end{aligned}$$

Transitions are labelled with atomic actions (*i.e.* multisets of basic action names).

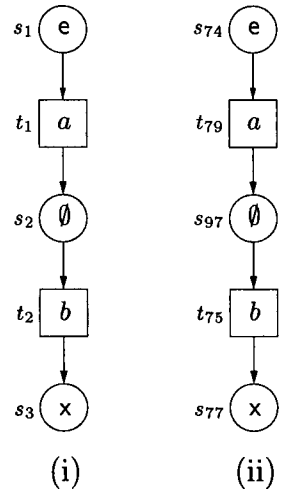


Figure 1.1: Labelled Petri nets

Figure 1.1 shows two nets that may be derived from the expression $E = a; b$. Net (i) is the graphical representation of the labelled net:

$$\begin{aligned}\Sigma \;=\; & (\{s_1, s_2, s_3\}, \{t_1, t_2\}, \{(s_1, t_1), (t_1, s_2), (s_2, t_2), (t_2, s_3)\}, \\ & \{(s_1, \mathsf{e}), (s_2, \emptyset), (s_3, \mathsf{x}), (t_1, \{a\}), (t_2, \{b\})\})\end{aligned}$$

Transitions and places are represented by rectangles and circles respectively. In giving a semantics to box expressions, it is both necessary and desirable to abstract away from place and transition names. This is reasonable since the choice of names for nodes in the net has no effect on either the structure or the behaviour of the net. The only purpose of such names is to give a means of identifying particular nodes. Hence, nets (i) and (ii) in Figure 1.1 are considered to be equivalent. It will be shown in Section 1.4 that such an abstraction away from node names amounts to defining a class of nets that are unique up to isomorphism. Therefore, place and transition names will usually be omitted where they are not required to illustrate a particular point.

1.3.3 Behaviour of labelled nets

Just as box expressions may be regarded as concurrent programs, so can labelled Petri nets. In this section, the process by which a labelled net is executed is described. The translation from an expression to a labelled net given in Section 1.3.5 is such that the execution of the net proceeds in a manner that matches the intended semantics described in Section 1.2.

During the execution of a labelled net, the current state of the system is recorded by a marking. Every place in the net may be marked by one or more tokens, and the sets of tokens in places determine the marking of the net. The initial marking (or state) of a labelled net is obtained by placing a single token in each entry place, and no tokens on any other place. The final marking of a net is reached when each exit place contains a single token, and every other place contains zero tokens. When the final marking is reached, the execution of the net has completed successfully.

The set of pre-places (post-places) of a transition is the set of places that have an arc to (from) that transition. The presence of tokens in the net may enable the execution of certain transitions. A transition is enabled if every pre-place contains sufficient tokens. The minimum number of tokens that each pre-place must contain is given by weight of the arc between that place and the transition. An enabled transition may be executed (or fired) by removing tokens from the pre-places and adding tokens in the post-places of the transition, where the number of tokens that are removed and added are determined by the arc weights. If several transitions are simultaneously enabled then it may be possible to execute them concurrently, or if they share common pre-places, there may be a choice between which transition is to be executed. A deadlocked state is reached, and the execution terminates when the current marking does not enable any transitions. If a deadlocked state is reached where the marking is not the final marking then the termination is unsuccessful.

The class of nets that can be derived from expressions over the syntax in Table 1.1 are such that during any execution of a net, no place will contain more than one token. An immediate corollary of this observation is that any transition that is connected by an arc with weight greater than one can never be enabled.

Figure 1.2 illustrates the execution of a net that has been obtained from the expression $a; b$. The initial marking enables the transition labelled a, which can fire by executing an a action to reach the state where only the internal place contains a token. The final marking is reached from this state by performing a b action. Once the final marking has been reached, there are no enabled transitions. Hence the execution terminates successfully. This behaviour can be seen to correspond with the intended semantics of the sequence operator.

In investigating the structural properties of nets in relation to the synthesis and axiomatisation problems, the behaviour and markings of nets do not need to be considered. Therefore, future diagrams that contain nets will not indicate

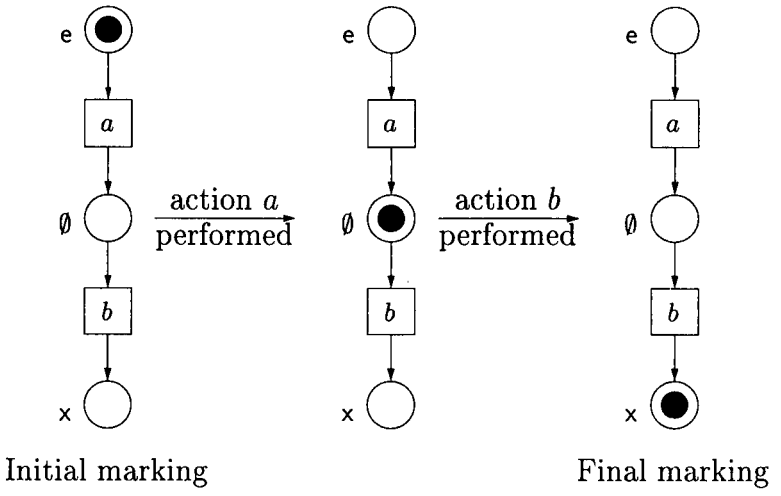


Figure 1.2: Behaviour of labelled nets

any marking – only the structure of the net will be shown.

1.3.4 Operations on labelled nets

Four operators on labelled nets, $\sqcup$, $\ominus$, $\otimes$ and $\oplus$ are used to implement the translation from box expressions to labelled nets. The first operator, net union ($\sqcup$) is only defined for disjoint nets. Two labelled nets, $\Sigma_1 = (S_1, T_1, W_1, \lambda_1)$ and $\Sigma_2 = (S_2, T_2, W_2, \lambda_2)$ are disjoint if $(S_1 \cup T_1) \cap (S_2 \cup T_2) = \emptyset$. The ability to generate disjoint nets relies on the abstraction away from place and transition names that was introduced in Section 1.3.2.

For disjoint nets, Σ_1 and Σ_2, the net union, $\Sigma_1 \sqcup \Sigma_2$ is defined by:

$$\Sigma_1 \sqcup \Sigma_2 = (S_1 \cup S_2, T_1 \cup T_2, W, \lambda_1 \cup \lambda_2)$$

Where $W : (S_1 \cup S_2 \cup T_1 \cup T_2) \times (S_1 \cup S_2 \cup T_1 \cup T_2) \rightarrow \mathcal{N}$ is given by:

$$W(n_1, n_2) = \begin{cases} W_1(n_1, n_2) & \text{if } n_1, n_2 \in S_1 \cup T_1 \\ W_2(n_1, n_2) & \text{if } n_1, n_2 \in S_2 \cup T_2 \\ 0 & \text{otherwise} \end{cases}$$

The $\ominus$ operator can be used to remove a set of nodes, N, from a net $\Sigma =$

(S, T, W, λ):

$$\Sigma \ominus N = (S - N, T - N, W\mid_{((S \cup T)-N)\times((S \cup T)-N)}, \lambda\mid_{(S \cup T)-N})$$

The place multiplication operator, $\otimes$, is used to create a new set of places from a collection of disjoint sets of existing places. For a net, $\Sigma = (S, T, W, \lambda)$, let $S_1, ..., S_k$ be non-empty, disjoint subsets of S. The set of new places, $S_1 \otimes ... \otimes S_k$ is defined by:

$$S_1 \otimes ... \otimes S_k = \{\{s_1, ..., s_k\} \mid s_i \in S_i \text{ for } 1 \leq i \leq k\}$$

A set of new places, created using the $\otimes$ operator can be added to the net using the $\oplus$ operator. Let $P = S_1 \otimes ... \otimes S_k$ be a set of new places, and $l \in \{\mathsf{e}, \emptyset, \mathsf{x}\}$ be the label which is to be assigned to each place in P. The net, $\Sigma \oplus (P, l)$, obtained by adding the set of new places to Σ is defined by:

$$\Sigma \oplus (P, l) = (S \cup P, T, W', \lambda')$$

where $W' : (S \cup P \cup T) \times (S \cup P \cup T) \to \mathcal{N}$, and λ' are defined as follows:

$$W'(n_1, n_2) = \begin{cases} W(n_1, n_2) & \text{if } n_1, n_2 \in S \cup T \\ \sum_{n \in n_1} W(n, n_2) & \text{if } n_1 \in P, n_2 \in S \cup T \\ \sum_{n \in n_2} W(n_1, n) & \text{if } n_1 \in S \cup T, n_2 \in P \\ 0 & \text{otherwise} \end{cases}$$

$$\lambda'(n) = \begin{cases} \lambda(n) & \text{if } n \in S \cup T \\ l & \text{if } n \in P \end{cases}$$

The $\oplus$ operator may also be used to add a set of new transitions to a net. Let X be a set of new transitions, where each new transition is a multiset of the set, T, of existing transitions in the net $\Sigma = (S, T, W, \lambda)$, and l be the labelling function which assigns a label to each new transition in X. The net $\Sigma \oplus (X, l)$, obtained by adding the set of new transitions to Σ, is defined by:

$$\Sigma \oplus (X, l) = (S, T \cup X, W', \lambda')$$

where $W' : (S \cup T \cup X) \times (S \cup T \cup X) \rightarrow \mathcal{N}$, and λ' are defined as follows:

$$W'(n_1, n_2) = \begin{cases} W(n_1, n_2) & \text{if } n_1, n_2 \in S \cup T \\ \sum_{n \in n_1} W(n, n_2) & \text{if } n_1 \in X, n_2 \in S \cup T \\ \sum_{n \in n_2} W(n_1, n) & \text{if } n_1 \in S \cup T, n_2 \in X \\ 0 & \text{otherwise} \end{cases}$$

$$\lambda'(n) = \begin{cases} \lambda(n) & \text{if } n \in S \cup T \\ l(n) & \text{if } n \in X \end{cases}$$

The addition of transitions to the net is more flexible than the addition of places as each transition in the set X added by $\Sigma \oplus (X, l)$ can be given a different label – *i.e.* l is a labelling function. In the case of addition of places, $\Sigma \oplus (P, l)$, l is a label that is common to every place $p \in P$.

1.3.5 Translation from expressions to nets

A Petri box is an equivalence class of labelled nets. The equivalence class is obtained by abstracting away from the place and transition names in the net, and it will be shown later that this corresponds to isomorphism[3]. For every box expression, E, there is a corresponding Petri box, denoted $\mathsf{box}(E)$. The compositional semantics of the box calculus describe a translation from box expressions to Petri boxes. This translation is achieved by associating a semantic rule with each syntactic operator in Table 1.1. The omission of place and transition names of labelled nets in diagrams corresponds to a representation of a Petri box. For such diagrams the representation can either be thought of as a class of structurally equivalent (isomorphic) nets, or as a particular net where the labelling of node names is omitted. Of course, it is easy to obtain the Petri box corresponding to a particular labelled net, and vice versa.

The semantic rules are compositional, which means that in constructing the Petri box for, for example $E_1 \,\square\, E_2$, the Petri boxes for E_1 and E_2 are

[3] [6] uses a weaker equivalence relation, duplication equivalence.

constructed, then combined using the semantic rule for choice composition. The important point is that the rule for choice composition works no matter how complex the expressions E_1 and E_2 are.

The semantics of the box expression syntax in Table 1.1 are implemented in terms of the operators on labelled nets: $\sqcup$, $\oplus$, $\ominus$ and $\otimes$ described in the previous section. In this Section, the semantics for atomic actions, parallel composition, choice composition, sequential composition and iteration are described, together with a semantics for the restriction and synchronisation operators, which in turn allow the semantics for expressions involving **stop** and scoping to be derived. The semantics for the remaining operators may be found in [6].

Figure 1.3 shows some example Petri boxes obtained from simple box expressions, illustrating the use of each of the operators whose semantics are described below. The parallel composition, choice, sequence and iteration semantic rules have a general form which consists of three components:

1. The union of a collection of nets, Σ_i for $1 \leq i \leq k$, for some k, is formed, using the $\sqcup$ operator. The nets Σ_i correspond to subexpressions of the expression being translated into a Petri box.

2. Sets of new places are created, using the $\otimes$ operator, applied to entry and exit interfaces of some of the nets, Σ_i, used in 1. These sets of places are added to the result of 1, using the $\oplus$ operator. The entry and exit interfaces of each Σ_i are used at most once in this step.

3. The original entry and exit interfaces used in 2 are removed from the result of 2, using the $\ominus$ operator.

The semantic rules for the synchronisation and restriction operators, use a set of transitions T^a, which is the set of transitions in T that have an a or $\hat{a}$ in their label. For a net, $\Sigma = (S, T, W, \lambda)$, T^a is defined by:

$$T^a = \{t \in T \mid \lambda(t) \cap \{a, \hat{a}\} \neq \emptyset\}$$

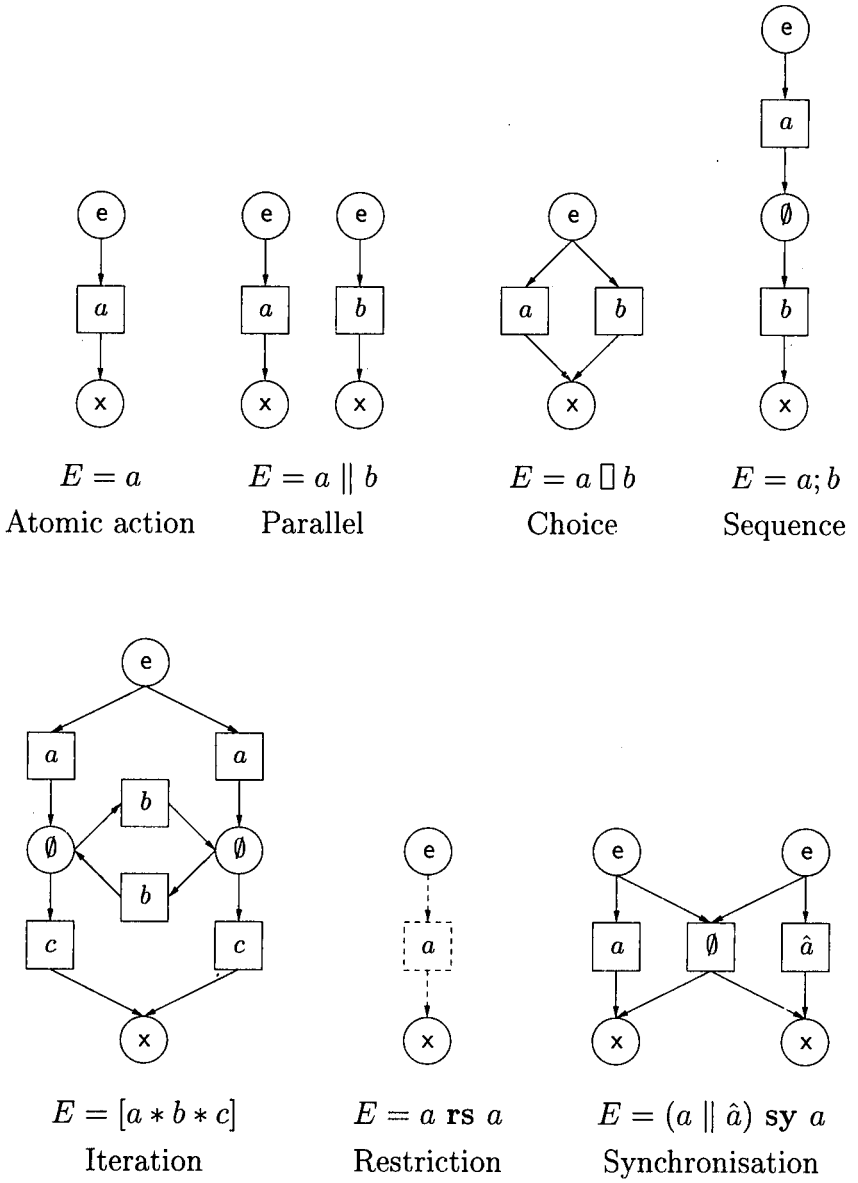


Figure 1.3: Box calculus semantics

In the following, $[\Sigma]$ denotes the class of nets equivalent to Σ when the abstraction away from place and transition names is made.

Atomic action

$$\mathsf{box}(\alpha) = [(\{s_1, s_2\}, \{t\}, \{(s_1, t_1), (t_1, s_2)\}, \{(s_1, \mathsf{e}), (s_2, \mathsf{x}), (t, \alpha)\})]$$

An atomic action is implemented by explicitly creating the Petri box given in Figure 1.3. The transition of the net is labelled with the multiset of basic actions that is the atomic action.

Parallel composition

$$\mathsf{box}(E_1 \parallel E_2) = \mathsf{box}(E_1) \parallel \mathsf{box}(E_2)$$

where, for $\Sigma_1 \in \mathsf{box}(E_1)$ and $\Sigma_2 \in \mathsf{box}(E_2)$ such that Σ_1 and Σ_2 are disjoint,

$$\mathsf{box}(E_1) \parallel \mathsf{box}(E_2) = [\Sigma_1 \sqcup \Sigma_2]$$

The net corresponding to the parallel composition of two expressions is constructed by taking the disjoint union of the nets for these expressions. The subnets are completely independent of each other, and can therefore execute concurrently.

Choice composition

$$\mathsf{box}(E_1 \,[]\, E_2) = \mathsf{box}(E_1) \,[]\, \mathsf{box}(E_2)$$

where, for $\Sigma_1 \in \mathsf{box}(E_1)$ and $\Sigma_2 \in \mathsf{box}(E_2)$ such that Σ_1 and Σ_2 are disjoint,

$$\begin{aligned}\mathsf{box}(E_1) \,[]\, \mathsf{box}(E_2) \;=\; [\Sigma_1 \sqcup \Sigma_2 &\oplus ({}^\bullet\Sigma_1 \otimes {}^\bullet\Sigma_2, \mathsf{e}) \\ &\oplus (\Sigma_1{}^\bullet \otimes \Sigma_2{}^\bullet, \mathsf{x}) \\ &\ominus ({}^\bullet\Sigma_1 \cup {}^\bullet\Sigma_2 \cup \Sigma_1{}^\bullet \cup \Sigma_2{}^\bullet)]\end{aligned}$$

When the choice composition of two expressions is taken, the entry and exit interfaces of the nets corresponding to these expressions are combined in the manner shown in Figure 1.3. In the example for choice composition in Figure 1.3, a token in the entry place can take one of two paths – executing either the "a" action, or the "b" action. This corresponds with the intended semantics of the choice operator.

Sequential composition

$$\mathsf{box}(E_1; E_2) = \mathsf{box}(E_1); \mathsf{box}(E_2)$$

where, for $\Sigma_1 \in \mathsf{box}(E_1)$ and $\Sigma_2 \in \mathsf{box}(E_2)$ such that Σ_1 and Σ_2 are disjoint,

$$\mathsf{box}(E_1); \mathsf{box}(E_2) = [\Sigma_1 \sqcup \Sigma_2 \oplus (\Sigma_1{}^\bullet \otimes {}^\bullet\Sigma_2, \emptyset) \ominus (\Sigma_1{}^\bullet \cup {}^\bullet\Sigma_2)]$$

When the sequential composition of two expressions is taken, the nets corresponding to these expressions are combined by joining the exit interface of the first net with the entry interface of the second net. The result is that the final marking of the first net is coincident with the initial marking of the second net – *i.e.* the second net does not begin execution until the execution of the first net has completed.

Iteration

$$\mathsf{box}([E_1 * E_2 * E_3]) = [\mathsf{box}(E_1) * \mathsf{box}(E_2) * \mathsf{box}(E_3)]$$

where, for $\Sigma_{11}, \Sigma_{12} \in \mathsf{box}(E_1)$, $\Sigma_{21}, \Sigma_{22} \in \mathsf{box}(E_2)$, and $\Sigma_{31}, \Sigma_{32} \in \mathsf{box}(E_3)$
such that $\Sigma_{11}, \Sigma_{12}, \Sigma_{21}, \Sigma_{22}, \Sigma_{31}$ and Σ_{32} are mutually disjoint,

$$[\mathsf{box}(E_1) * \mathsf{box}(E_2) * \mathsf{box}(E_3)] = [\Sigma_{11} \sqcup \Sigma_{12} \sqcup \Sigma_{21} \sqcup \Sigma_{22} \sqcup \Sigma_{31} \sqcup \Sigma_{32} \oplus ({}^\bullet\Sigma_{11} \otimes {}^\bullet\Sigma_{12}, \mathsf{e})$$

$$\oplus\,(\Sigma_{31}{}^{\bullet} \otimes \Sigma_{32}{}^{\bullet}, \mathsf{x})$$
$$\ominus\,({}^{\bullet}\Sigma_{11} \cup {}^{\bullet}\Sigma_{12} \cup \Sigma_{31}{}^{\bullet} \cup \Sigma_{32}{}^{\bullet})$$
$$\oplus\,(\Sigma_{11}{}^{\bullet} \otimes {}^{\bullet}\Sigma_{21} \otimes \Sigma_{22}{}^{\bullet} \otimes {}^{\bullet}\Sigma_{31}, \emptyset)$$
$$\oplus\,(\Sigma_{12}{}^{\bullet} \otimes {}^{\bullet}\Sigma_{22} \otimes \Sigma_{21}{}^{\bullet} \otimes {}^{\bullet}\Sigma_{32}, \emptyset)$$
$$\ominus\,(\Sigma_{11}{}^{\bullet} \cup {}^{\bullet}\Sigma_{21} \cup \Sigma_{22}{}^{\bullet} \cup {}^{\bullet}\Sigma_{31}$$
$$\cup\,\Sigma_{12}{}^{\bullet} \cup {}^{\bullet}\Sigma_{22} \cup \Sigma_{21}{}^{\bullet} \cup {}^{\bullet}\Sigma_{32})]$$

When the iteration operator $[E_1 * E_2 * E_3]$ is used the nets corresponding to the three expressions are combined as shown in the example in Figure 1.3. Notice that two copies of each net are used in this construction – this is to ensure that the resulting net is pure, which means there is no pair of nodes n_1 and n_2 such that there are arcs both from n_1 to n_2 and from n_2 to n_1. If only a single copy of the nets are used, then, for example, the implementation of $[a * b * c]$ would not be pure because the transition labelled b would have an arc both to and from the same place. In fact, only two copies of the nets corresponding to E_2 and E_3 are required to ensure that the construction is pure. A second copy of the net corresponding to E_1 is included to make the net (and the set of reachable states) symmetrical, allowing Petri net based verification algorithms to more easily detect the redundancy in the implementation of iteration expressions. At the initial marking of an iteration net, there are two possible paths of execution. The behaviour of the net will be the same whichever path is taken. In the example net, note that there is a cycle consisting of the two copies of the net corresponding to the expression "b". Once an a transition has been executed, b transitions can be executed any number of times (including zero), before one of the c transitions is executed and the final marking is reached. This corresponds to the intended semantics of iteration expressions.

Restriction

$$\mathsf{box}(E \textbf{ rs } a) = \mathsf{box}(E) \textbf{ rs } a$$

where, for $\Sigma = (S, T, W, \lambda) \in \mathsf{box}(E)$,

$$\mathsf{box}(E)\ \mathbf{rs}\ a\ =\ [\Sigma \ominus T^a]$$

The restriction of an expression by a basic action name is achieved by removing every transition that has a label containing that action name or its conjugate, from the net corresponding to that expression. This results in certain paths of the execution being blocked, and means that the final marking of the net may no longer be reachable. The restriction operator is generally used in conjunction with synchronisation, in such a way that the final marking remains reachable.

Synchronisation

$$\mathsf{box}(E\ \mathbf{sy}\ a) = \mathsf{box}(E)\ \mathbf{sy}\ a$$

where, for $\Sigma = (S, T, W, \lambda) \in \mathsf{box}(E)$, let τ, be a finite multiset of the set of transitions, T^a, such that τ contains at least two elements. τ is a valid synchronisation if and only if the multiset sum of the labels of the transitions in τ contains at least $|\tau| - 1$ a and $\widehat{a}$ basic actions. Hence, the set of new transitions created by synchronising Σ by a basic action, a is given by:

$$T_{sy} = \{\tau \mid |\tau| \geq 2 \wedge \min(\sum_{t \in \tau} \lambda(t)(a), \sum_{t \in \tau} \lambda(t)(\widehat{a})) \geq |\tau| - 1\}$$

The labelling function, l defines the label of τ to be the multiset sum of the labels of the transitions in τ minus $|\tau| - 1$ copies of the a and $\widehat{a}$ basic actions:

$$l(\tau) = (\sum_{t \in \tau} \lambda(t)) - ((|\tau| - 1) \cdot \{a, \widehat{a}\})$$

The synchronisation operation on Petri boxes is defined by:

$$\mathsf{box}(E)\ \mathbf{sy}\ a\ =\ [\Sigma \oplus (T_{sy}, l)]$$

The semantics for synchronisation presented here are slightly different from those in [6], where the condition requiring $|\tau| \geq 2$ is not imposed. Hence, the semantics in [6] create a duplicate of every transition in the set T^a. While these duplicate transitions are not significant for the duplication equivalence of [6], they are for the stronger equivalence of isomorphism investigated here.

The synchronisation of an expression by a basic action, a, is achieved by adding new transitions to the net corresponding to the expression being synchronised. A new transition is added for every pair of synchronising transitions – *i.e.* a pair of transitions, with one transition label containing the synchronising action, a, and the other containing the conjugate of the synchronising action, $\widehat{a}$. Each new transition inherits the arcs from the pair of synchronising transitions, and is labelled with the multiset union of the labels of the synchronising transitions, minus the two basic actions, a and $\widehat{a}$ that contributed to the synchronisation. For example, the $\emptyset$ transition in the net in Figure 1.3 arises from the synchronisation of the a and $\hat{a}$ transitions.

A labelled net, Σ is called an implementation of a box expression, E, if $\Sigma \in \mathsf{box}(E)$. Note that:

$$\Sigma \in \mathsf{box}(E) \Leftrightarrow [\Sigma] = \mathsf{box}(E)$$

An implementation of an expression, E can be constructed from disjoint implementations of the subexpressions of E. For example, an implementation, Σ, of $E = E_1; E_2$, can be constructed from disjoint implementations, Σ_1 and Σ_2 of E_1 and E_2, using:

$$\begin{aligned} \Sigma \;=\; & \Sigma_1 \sqcup \Sigma_2 \oplus (\Sigma_1{}^\bullet \otimes {}^\bullet\Sigma_2, \emptyset) \\ & \ominus (\Sigma_1{}^\bullet \cup {}^\bullet\Sigma_2) \end{aligned}$$

A similar procedure can be used for the other operators in Table 1.1.

1.4 Equivalence of expressions and nets

One of the motivations for the design of the Petri Box Calculus was the ability to define notions of equivalence in a process algebra framework that were traditionally only found in the domain of Petri nets. In this section, it is shown how the equivalence of box expressions can be based on an equivalence for labelled nets. There are many candidates for the equivalence of Petri nets. An idea of the range of possible equivalences can be obtained from [45]. Two structural equivalences, isomorphism and duplication (renaming) equivalence, are formally defined in this section. These notions are used in the remainder of the thesis as a basis for the investigation into the synthesis problem and the production of an axiomatisation. Structural, rather than behavioural equivalences are chosen for the investigation because the synthesis and axiomatisation problems are simpler, and the work should provide a basis for an investigation into behavioural equivalences. Chapter 6 discusses an approach to the synthesis problem for behavioural equivalences, that is based on structural analysis similar to those used for isomorphism and duplication equivalence. Structural equivalences are much stronger than behavioural equivalences, and any reasonable notion of behavioural equivalence will encompass equivalences such as isomorphism and duplication equivalence. Therefore, it should be expected that an axiomatisation for a structural equivalence could form the basis of an axiomatisation of a behavioural equivalence, while the reverse is not true.

For an equivalence relation, $=_n$, over nets, a corresponding equivalence $=_e$ can be defined over box expressions as follows:

$$E_1 =_e E_2 \Leftrightarrow \Sigma_1 =_n \Sigma_2 \text{ where } \Sigma_1 \in \mathsf{box}(E_1) \text{ and } \Sigma_2 \in \mathsf{box}(E_2)$$

This definition of equivalence for box expressions requires that the equivalence relation, $=_n$, encompasses isomorphism – *i.e.* if the nets Σ_1 and Σ_2 are isomorphic, then $\Sigma_1 =_n \Sigma_2$. This is a reasonable assumption, because isomorphism simply abstracts away from node names, which have no effect on the semantics of the net. Any reasonable notion of equivalence should be expected to

preserve the abstraction.

The notation $[E]_n$ is used to represent the class of nets equivalent, according to the relation $=_n$, to an arbitrary member of the Petri box constructed for E. When $=_n$ corresponds to isomorphism, then $[E]_n$ and $\mathsf{box}(E)$ describe the same class of labelled nets. For any other reasonable notion of equivalence, $=_n$, $\mathsf{box}(E) \subseteq [E]_n$, which amounts to saying that $=_n$ encompasses isomorphism.

Two equivalence relations based on the structure of labelled nets are considered. The stronger equivalence is isomorphism, which corresponds to the abstraction away from place and transition names. Duplication equivalence, in addition, abstracts away from the duplication of nodes in the net. Writing $\Sigma_1 =_{iso} \Sigma_2$ ($\Sigma_1 =_{dup} \Sigma_2$) indicates that the nets Σ_1 and Σ_2 are isomorphic (duplication equivalent). Similarly, $E_1 =_{iso} E_2$ ($E_1 =_{dup} E_2$) indicates that E_1 is equivalent to E_2 according to net isomorphism (duplication equivalence). However, if it is clear what notion of equivalence is being used, the $=$ symbol will often be used in place of $=_{iso}$ or $=_{dup}$.

The motivation for investigating the synthesis and axiomatisation problems for net semantic isomorphism is that for any expression, E, the class of nets $[E]_{iso}$ is identical to that described by $\mathsf{box}(E)$. This correspondence significantly simplifies the synthesis problem. For equivalence relations weaker than isomorphism, members of the equivalence class of boxes will only be equivalent, under that relation, to the implementation of some box expression. They may not necessarily be implementations of a box expression themselves. The investigation into duplication equivalence gives an idea of the additional problems that are encountered when the correspondence between equivalent nets, and Petri boxes is weakened.

1.4.1 Isomorphism

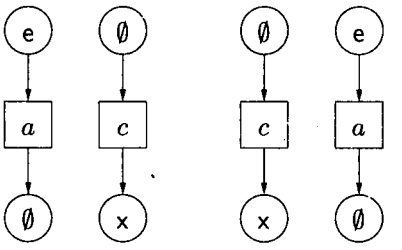
Isomorphism is possibly the strongest equivalence relation over nets (apart from identity). Two nets are isomorphic if there is a one-to-one correspondence between the nodes that preserves adjacency and node labelling. Formally, the

nets $\Sigma_1 = (S_1, T_1, W_1, \lambda_1)$ and $\Sigma_2 = (S_2, T_2, W_2, \lambda_2)$ are isomorphic if there exists a sort-preserving bijection $\rho : (S_1 \cup T_1) \to (S_2 \cup T_2)$ such that:

$$\forall n_1, n_2 \in S_1 \cup T_1 \quad : \quad W_1(n_1, n_2) = W_2(\rho(n_1), \rho(n_2))$$
$$\forall n \in S_1 \cup T_1 \quad : \quad \lambda_1(n) = \lambda_2(\rho(n))$$

$[\Sigma]_{iso}$, which is the same as $[\Sigma]$, is used to denote the class of labelled nets that are isomorphic to Σ. If $\Sigma_1 =_{iso} \Sigma_2$, then the sets of nodes $n_1 \subseteq S_1 \cup T_1$, $n_2 \subseteq S_2 \cup T_2$ are isomorphic, denoted $n_1 =_{iso} n_2$ if there exists an isomorphism, ρ for Σ_1 and Σ_2, such that $n_2 = \{\rho(n) \mid n \in n_1\}$. This can be extended to the isomorphism of sets of sets of nodes in the obvious way.

Under isomorphism, there is an epimorphism (many-to-one, onto relation) from box expressions to labelled nets – *i.e.* there may be several expressions that produce isomorphic nets. For example, the implementations of $((a;b);c)$ **rs** b and $(a;(d;c))$ **rs** d are isomorphic, as shown in Figure 1.4.



$(a;(b;c))$ **rs** b $\quad$ $(a;(d;c))$ **rs** d

Figure 1.4: Isomorphic nets

1.4.2 Duplication equivalence

Duplication (or renaming) equivalence is the equivalence relation defined and used in [6] to correspond to the class of nets that is a Petri box. Duplication equivalence is weaker than isomorphism:

$$\Sigma_1 \text{ and } \Sigma_2 \text{ isomorphic } \Rightarrow \Sigma_1 \text{ and } \Sigma_2 \text{ duplication equivalent}$$

but:

$$\Sigma_1 \text{ and } \Sigma_2 \text{ duplication equivalent } \not\Rightarrow \Sigma_1 \text{ and } \Sigma_2 \text{ isomorphic}$$

Duplication equivalence is based on an equivalence relation over the elements of a labelled net. In a net $\Sigma = (S, T, W, \lambda)$, two nodes $n_1, n_2 \in S \cup T$ are duplicates of each other, written $n_1 =_{dup} n_2$, if $\lambda(n_1) = \lambda(n_2)$ and for all $n \in S \cup T$, both $W(n_1, n) = W(n_2, n)$ and $W(n, n_1) = W(n, n_2)$. Duplication equivalence is defined by extending the relation "to duplicate each other" to an equivalence relation on labelled nets. Formally, the nets $\Sigma_1 = (S_1, T_1, W_1, \lambda_1)$ and $\Sigma_2 = (S_2, T_2, W_2, \lambda_2)$ are duplication equivalent if there exists a relation $\rho \subseteq (S_1 \cup T_1) \times (S_2 \cup T_2)$ such that:

- The relation is surjective (onto), and sort-preserving on places and transitions – *i.e.* $\rho(S_1) = S_2$, $\rho^{-1}(S_2) = S_1$, $\rho(T_1) = T_2$, and $\rho^{-1}(T_2) = T_1$.
- The relation preserves arcs, and arc weights – *i.e.* if $(s_1, s_2) \in \rho$ and $(t_1, t_2) \in \rho$, then $W(s_1, t_1) = W(s_2, t_2)$ and $W(t_1, s_1) = W(t_2, s_2)$.
- The relation preserves labels – *i.e.* if $(n_1, n_2) \in \rho$, then $\lambda(n_1) = \lambda(n_2)$.
- The relation is injective (one-to-one) on classes of duplicate nodes – *i.e.* if $(n_1, n_2) \in \rho$, and $(m_1, m_2) \in \rho$ then $n_1 =_{dup} m_1$ in Σ_1 if and only if $n_2 =_{dup} m_2$ in Σ_2.
- The relation is full for classes of duplicate nodes – *i.e.* if $n_1 =_{dup} m_1$ in Σ_1 and $n_2 =_{dup} m_2$ in Σ_2 then $(n_1, n_2) \in \rho$ if and only if $(m_1, m_2) \in \rho$.

$[\Sigma]_{dup}$ is used to denote the class of labelled nets that are duplication equivalent to Σ.

A canonical representative, up to isomorphism, of a class of duplication equivalent nets can be obtained by removing all but one of each set of duplicate nodes. Hence, two nets are duplication equivalent if and only if their canonical representatives are isomorphic. In Figure 1.5, net (i) shows the implementation of:

$$E = (((a \parallel b) \,\square\, (a \parallel b)) \parallel ((a \parallel b) \,\square\, (a \parallel b))) \textbf{ rs } b$$

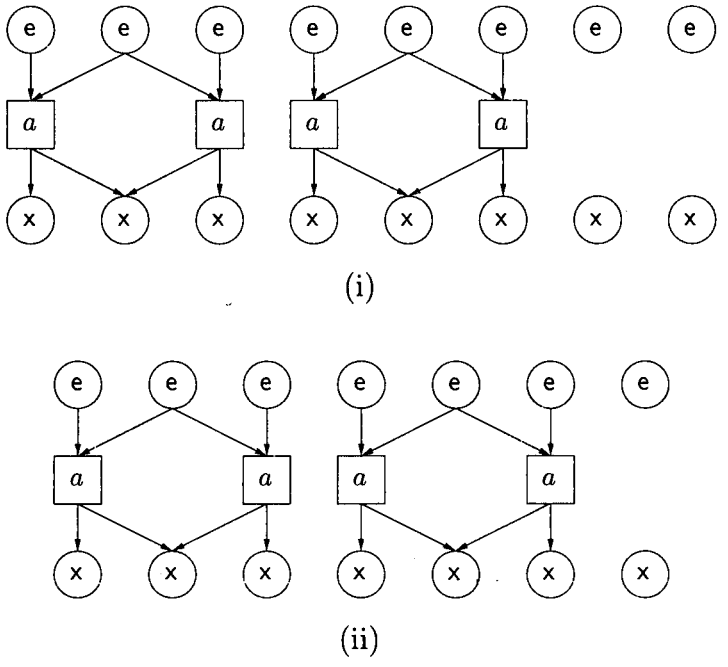


Figure 1.5: Duplication equivalent nets

and net (ii) shows the canonical representative for net (i). Although there is a strong relationship between isomorphism and duplication equivalence, Figure 1.5 illustrates a possible source of difficulty in extending a synthesis algorithm and axiomatisation for isomorphism to one for duplication equivalence. While net (i) is the implementation of a box expression, there is no box expression with an implementation isomorphic to net (ii). Even worse, net (ii) is the natural choice for a canonical representative for net (i). This problem is discussed further in Chapter 5.

1.5 Synthesis and axiomatisation problems

The synthesis problem is to provide an algorithmic translation from labelled nets to box expressions – *i.e.* given an arbitrary labelled net as input, try to synthesise an expression whose implementation is equivalent (under some particular equivalence relation) to the input net. For a particular notion of equivalence, $=_n$, the decision problem associated with the synthesis problem

can be stated as:

> BOX EXPRESSION SYNTHESIS
> INSTANCE: Labelled net, Σ.
> QUESTION: Does there exist a box expression, E,
> such that $\mathsf{box}(E) = [\Sigma]_n$.

The problem may be simplified slightly by allowing only those inputs for which there exists a solution. This restriction does not affect the time complexity of the problem. For a particular notion of equivalence, $=_n$, the synthesis problem can be restated thus:

> BOX EXPRESSION SYNTHESIS
> INSTANCE: Net, Σ, for which there exists a synthesisable expression.
> SOLUTION: Box expression, E, such that $\mathsf{box}(E) = [\Sigma]_n$.

The motivation for investigating the synthesis problem is twofold: Firstly, the investigation provides a detailed analysis of the semantics of box expressions, allowing an axiomatisation of the box algebra to be obtained, and secondly, a solution to the synthesis problem allows process algebraic representations to be derived for a class of Petri nets. While the class of nets for which an expression can be synthesised is small for structural equivalences, it will be much larger, perhaps even covering the entire class of Petri nets, for behavioural equivalences. Bearing this in mind, a synthesis algorithm for behavioural equivalences is a much more attractive proposition, and could have immense practical applications. An initial investigation into isomorphism and duplication equivalence will hopefully provide insight into, and a solid basis for the extension to behavioural equivalences. Bearing these motivations in mind, there are two desirable properties that a synthesis algorithm may have:

- **Determinism:** Given two equivalent nets as input, the algorithm should synthesise identical expressions. This means that the synthesis algorithm implicitly defines a canonical form for the equivalence classes of box expressions. Designing an algorithm which has the determinism property

will not be trivial because there is a many-to-one relation between box expressions and classes of Petri boxes, even under the strongest equivalence relation, isomorphism. The determinism property is useful from a theoretical point of view, in that it allows the synthesis algorithm to be used to show that an axiomatisation of the Petri Box Calculus is complete. While a non-deterministic algorithm can still be of use in proving completeness, the task is significantly easier using a completely deterministic synthesis algorithm. Section 2.4 in Chapter 2 discusses in detail the means by which an analysis of a synthesis algorithm can be used to derive an axiomatisation.

- **Efficiency:** The time complexity of the synthesis algorithm should be, at most, polynomial in the size of the input net. An efficient algorithm for synthesis would give more scope for practical applications of the algorithm. While a solution to the synthesis problem for structural equivalences such as isomorphism and duplication equivalence has limited practical use, the investigation provides a basis for the extension to more reasonable (from a practical point of view) net equivalences. The extension to behavioural equivalences is discussed in Chapter 6. The efficiency of the synthesis algorithm has no effect on the possibility of deriving an axiomatisation, although an efficient synthesis algorithm may lead to efficient procedures for the automatic generation of proofs, using the axiomatisation. In this respect, efficiency should be regarded as secondary to minimising the amount of non-determinism in the algorithm.

It is not imperative that the expression synthesised from a particular input net is unique. There will often be a tradeoff between the amount of determinism in the algorithm and the efficiency of the algorithm. An analysis of the points of non-determinism in the synthesis algorithm, may still provide a means for defining a canonical form for box expressions.

The problem of finding an axiomatisation is not an algorithmic problem. For a particular notion of equivalence $=_n$ over labelled nets, an axiomatisation characterises the corresponding equivalence over expressions, $=_e$, by means of a set of axioms (rewriting rules). It is important for an axiomatisation to be both sound and complete. For an axiom system to be sound, whenever the system equates two expressions, E_1 and E_2, then it is true that $\Sigma_1 =_n \Sigma_2$, where Σ_1 and Σ_2 are implementation of E_1 and E_2. An axiom system is complete if, for every pair of expressions E_1 and E_2 such that $E_1 =_e E_2$, it is possible to show the equivalence of E_1 and E_2 by applying axioms.

As an example, a sound, but not complete axiom system for net semantic isomorphism is presented, and used to show that the expressions $((a;b);c)$ **rs** b and $(a;(d;c))$ **rs** d are equivalent. The axiom system consists of three axioms, dealing respectively with associativity of sequential composition, propagation of the restriction operator, and restriction of atomic actions.

$$E_1;(E_2;E_3) = (E_1;E_2);E_3 \qquad \texttt{AXIOM 1}$$

$$(E_1;E_2)\ \mathbf{rs}\ a = (E_1\ \mathbf{rs}\ a);(E_2\ \mathbf{rs}\ a) \qquad \texttt{AXIOM 2}$$

$$\alpha\ \mathbf{rs}\ a = \begin{cases} \mathbf{stop} & \text{if } a \in \alpha \\ \alpha & \text{otherwise} \end{cases} \qquad \texttt{AXIOM 3}$$

Implementations of $E_1 = ((a;b);c)$ **rs** b and $E_2 = (a;(d;c))$ **rs** d are shown in Figure 1.4, where it can be seen that they are isomorphic, and hence $E_1 =_{iso} E_2$. A proof that $E_1 =_{iso} E_2$ is possible, by applying the three axioms as

follows:

$$
\begin{aligned}
((a;b);c)\ \mathbf{rs}\ b &= ((a;b)\ \mathbf{rs}\ b);(c\ \mathbf{rs}\ b) && \text{by AXIOM 2}\\
&= ((a\ \mathbf{rs}\ b);(b\ \mathbf{rs}\ b));(c\ \mathbf{rs}\ b) && \text{by AXIOM 2}\\
&= (a;(b\ \mathbf{rs}\ b));c && \text{by AXIOM 3}\\
&= (a;\mathbf{stop});c && \text{by AXIOM 3}\\
&= a;(\mathbf{stop};c) && \text{by AXIOM 1}\\
&= a;((d\ \mathbf{rs}\ d);c) && \text{by AXIOM 3}\\
&= (a\ \mathbf{rs}\ d);((d\ \mathbf{rs}\ d);(c\ \mathbf{rs}\ d)) && \text{by AXIOM 3}\\
&= (a\ \mathbf{rs}\ d);((d;c)\ \mathbf{rs}\ d) && \text{by AXIOM 2}\\
&= (a;(d;c))\ \mathbf{rs}\ d && \text{by AXIOM 2}
\end{aligned}
$$

As well as characterising an existing notion of equivalence from the domain of Petri nets using an axiom system, it is possible to define new notions of equivalence by introducing a set of axioms that are used to determine whether expressions are equivalent. However, this possibility is not considered further.

1.6 Related work

In this section, work relating to the Petri Box Calculus is described together with some of the work on the synthesis of process algebraic terms from Petri nets, and the axiomatisation of process algebras.

1.6.1 The Petri Box Calculus

The main theoretical results relating to the Petri Box Calculus are brought together in a practical form in the PEP (Programming Environment based on Petri Nets) tool [10]. PEP is a simulation, modelling and verification tool that can work with $B(PN)^2$ (Basic Petri Net Programming Notation) programs, high level and elementary Petri Box expressions, and high level and elementary Petri nets. Figure 1.6 illustrates the relationship between the various components of PEP.

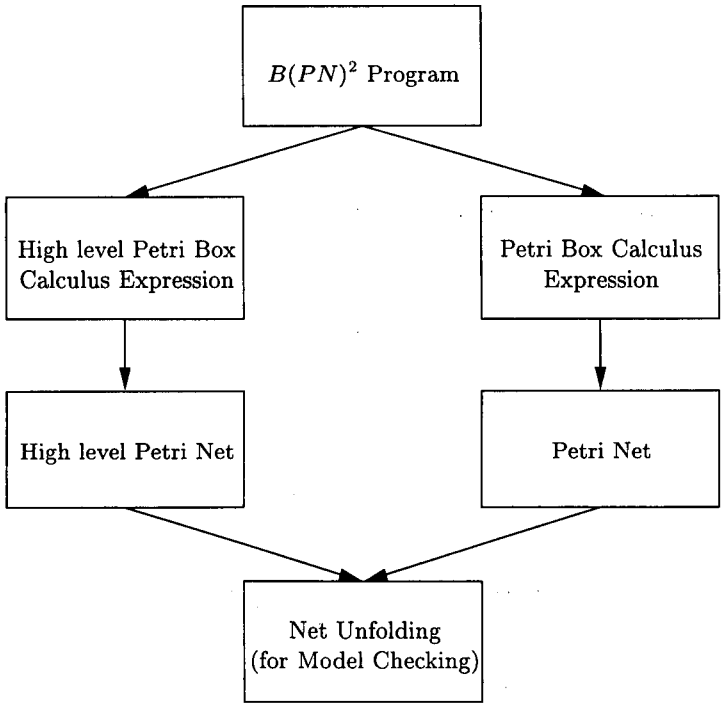


Figure 1.6: Components of the PEP tool

$B(PN)^2$ [7] is a high level concurrent programming language with support for various types including stacks and queues, and programming constructs such as loops and procedures. The semantics of $B(PN)^2$ programs are given in terms of Petri Box Expressions. PEP allows a $B(PN)^2$ program to be translated to a high level Box expression [9, 22], or to an elementary Petri Box Expression [7]. High level and elementary Petri Box expressions are essentially the same, except that actions in high level box expressions have values associated with them.

The Petri Box Expression may then be translated into a corresponding net [8, 5]. The high level, or M-net, representation of a program is generally less complex structurally, and is much more suited to representing data types and operations on variables than the elementary net representation of a program [22]. The Petri net is concrete representation of the semantics of a $B(PN)^2$ program, and can be used to simulate the execution of the program. PEP has the facility to relate the firing of actions in the net directly back to the

execution of statements in the $B(PN)^2$ program. The work on operational semantics for the Box Algebra, [36, 37, 35] provides the basis for a similar relationship between the execution of Box Expressions and the execution of statements in the $B(PN)^2$ program.

The model checking algorithm in PEP is based around the work in [20], where a representation of the unfolding is constructed from the Petri Net. The validity of logical statements about the net/Box expression/$B(PN)^2$ program may be checked using the unfolding. The specification for a concurrent system may be given in terms of properties that such a system must satisfy. The PEP tool could be used to model a system at the level of a $B(PN)^2$ program, a Petri Box Expression, or even a Petri net, and the model checker used to verify that the model satisfies the specification.

The remainder of this section considers how any work on a synthesis algorithm or axiomatisation for the Petri Box Calculus would fit into the framework illustrated in Figure 1.6. At the moment, it is only possible to go from a high level representation of a system ($B(PN)^2$ program or Box Expression) to a lower level one. A synthesis algorithm provides a translation from a Petri net to a Petri Box Expression. This would give the flexibility of being able to work at any level, and may give the scope to design concurrent systems at the net level, or reuse existing net based designs, and synthesise them to higher level and simpler representations in the form of Box Expressions, or even $B(PN)^2$ programs.

Recall that both the synthesis algorithm and axiomatisation problems are predicated on a net equivalence. While the class of nets that are equivalent to implementations of Box Expressions is relatively small for structural semantics such as isomorphism and duplication equivalence, a much larger class of nets will be synthesisable for behavioural net semantics. This is borne out by the fact that the class of Petri Boxes is expressive enough to encode the semantics of high level languages such as $B(PN)^2$ and OCCAM [7, 30, 31]. The implication is that for a particular behavioural net equivalence, it will be possible

to synthesise an expression for any net whose behaviour can be described by a $B(PN)^2$ or OCCAM program. It is relatively easy to show that if the net equivalence is weak enough, for example string equivalence [32, 45], then the entire class of Petri nets is synthesisable to Box expressions.

The benefits of an axiomatisation for the Box Algebra are that it would then be possible to manipulate a model for a concurrent system at the Box expression level, with a corresponding effect at the net and $B(PN)^2$ level. For example, such manipulations may be carried out to optimise a system. The advantage of using axioms to perform the manipulations is that the behaviour of the system is guaranteed to be preserved (for whichever net semantic the axioms system is based on). The result is that if manipulations are carried out on a system that has already been model checked, then it will not be necessary to recheck the system after the manipulations.

1.6.2 Synthesis of terms from nets

The Petri Box Calculus is unusual in that the semantics of the algebra of Box expressions is defined in terms of a class of Petri nets. Generally, the semantics of process algebras are originally defined in some other way, although there has been several pieces of work on giving net semantics to CCS and TCSP [21, 24, 26, 43, 44, 48]. Since this work can be considered an extension of the original work on CCS and TCSP [28, 41], it is perhaps unsurprising that there has been little or no work on the synthesis of process algebraic terms from the classes of nets described by these semantics. Instead, the majority of the work on the synthesis of terms from nets has considered the problem for the class of all place transition nets [1, 14, 16, 17, 18].

In [14], Boudol, Roucairol and De Simone synthesise a term consisting of a large number of parallel components corresponding to the places and transitions in the net. The arcs of the net are encoded by the communication capabilities of the parallel components, which represent the ability to pass a token from one place to another. The term representation is a very low level

representation of the net, and corresponds almost directly to the structure of the net.

In comparison, Dietz and Schrieber [18] and Christensen [17] base the synthesis on behavioural equivalences, bisimulation equivalence and branching process equivalence, and construct terms where the parallel components represent parallelism in the behaviour, rather than the structure of the net. The term semantics is close to the the idea of processes [46], and the branching processes of a net [19]. Unlike the usual infinite lattice branching process representation of [19], the term representation constructed by [18] is finite given a finite place transition net. The work in [18] restricts the class of input nets to those with binary synchronisation and binary choice. The synthesis process is based around the following steps:

- An intermediate "synchronisation free" (SF) net representation is used, where every transition has at most one incoming arc. A modified firing rule which encodes information about the synchronisations in the original net is applied to the SF net, so that an arbitrary place transition net is bisimilar to its synchronisation free version with the modified action firing rule.

- The term representation is derived from the synchronisation free net, and the reachable markings (under the modified firing rule) of that net.

- Synchronisation in the term representation is enforced by means of restriction. This is shown to be equivalent to the step from the standard firing rule to the modified firing rule in the SF net. It follows that the restricted term representation is bisimilar to the original input net.

The work by Baeten and Bergstra [1] and Basten and Voorhoeve [16] is more similar to that of [14]. The term representation of [16] is slightly different to other work in that actions in the algebraic term do not correspond to transitions in the net. Instead, actions correspond to the production and

consumption of tokens in the net. Basten and Voorhoeve synthesise an ACP [2] style term representation called PTNA (Place/Transition Net Algebra) for an arbitrary place transition net. An operational semantics for PTNA is introduced such that the algebraic semantics is consistent with the interleaving semantics of the net from which the PTNA term is synthesised. This work easily extends to high level Petri nets, provided the range of values of tokens is finite.

The synthesis problem for the Petri Box Calculus is made simpler by the fact that it is only necessary to consider a limited class of nets as input to the problem – *i.e.* the class of nets that may be derived from Box expressions. It appears that the problem will be simpler for structural equivalences such as isomorphism and duplication equivalence due to the fact that the reachable markings of the net do not need to be considered. It seems most likely that the synthesis process will produce terms where the actions in the algebraic term correspond to transitions in the input net (rather than the representation used by [16]), as this is way in which the semantics for Box expressions are currently defined.

1.6.3 Axiomatisation of Process Algebra

The aim of relating the synthesis and axiomatisation problems for the Petri Box Calculus is to create a framework in which results for a particular subset of the calculus, and for a particular net equivalence can easily be reused for different subsets/net semantics.

Work on axiomatisation of process algebras, such as CCS and CSP, [41, 28, 23, 11] is generally for behavioural semantics such as observation equivalence, or bisimulation. The most difficult part of any axiomatisation is showing that the axiom system is complete. The completeness proof for the axiomatisation of observation congruence in [42] is based around the following steps [23]:

1. It is shown that any expression can be rewritten in a form where the

expression is guarded.

2. A standard set of equations relating to guarded expressions is presented, and it is shown that any guarded expression satisfies these equations.

3. Any standard set of equations can be converted into a saturated one, while preserving the property of being provably satisfied by an expression.

4. Two congruent processes that each provably satisfy a saturated standard guarded set of equations, provably satisfy a common guarded set of equations.

5. Finally, if two guarded expressions satisfy the same guarded set of equations, then they are provably equal.

In moving from the domain of observation congruence [41] to branching bisimulation [27], only those steps above that rely purely on axioms for strong congruence (which is stronger than both observation congruence and branching congruence) may be reused. The remainder of the proof in [23] had to be reworked for the new semantics.

What is interesting that Glabbeek proved the completeness theorem for branching congruence on recursion free process expressions in two different ways. In [23] a proof specific to that particular problem is presented. In [27], a proof based on graph transformations [11] is used. While the proof in [23] is much shorter, the graph transformation method used in [27] is more generic, and allows completeness proofs to be generated for arbitrary interleaving equivalences with little effort.

The lessons that can be learned for the investigation into the axiomatisation problem for the Petri Box Calculus are that it is perhaps easier to move from a stronger equivalence to a weaker one than the other way round. If this approach is used, then the axioms and proofs for the stronger equivalence can be reused for the weaker equivalence. The only work that then needs to be

done is to capture the essence of the difference between the two notions of equivalence. Also, while a generic proof technique or framework may involve more work for a single subset of the calculus or single net semantic, that work will be worthwhile if the results can be reused for different subsets of the calculus and different net semantics.

1.7 Summary

Chapter 2 describes possible approaches to solving the synthesis problem, and gives details of the relationship between synthesis and the problem of finding an axiomatisation. Some other problems, such as checking equivalence of expressions and nets, and generating a proof of equivalence are introduced, and their relationship to the synthesis problem in terms of runtime complexity discussed. Chapter 2 concludes by investigating various properties of nets, useful for the analysis required for a solution to the synthesis problem. Chapter 3 presents a solution to the synthesis problem for a basic subset of the Petri Box Calculus. The algorithm described is shown to be correct, and an axiomatisation is derived. Solutions to some of the related problems introduced in Chapter 2 are also presented.

Chapter 4 extends the investigation to a subset of the Petri Box Calculus that contains the synchronisation operator. The synthesis problem is shown to be NP-hard. However, by using a more expressive syntax to represent the synthesised expression, a polynomial time solution can be given. As with Chapter 3, the synthesis algorithm is shown to be correct, an axiomatisation is derived, and the related problems are discussed.

In Chapter 5, the axiom systems derived for the basic subset of the calculus in Chapter 3, and the basic calculus with synchronisation operator in Chapter 4 are extended from isomorphism to duplication equivalence. While extending the axiomatisation for the basic calculus requires relatively little work, a different approach to that in Chapter 4 is taken to produce an ax-

iomatisation for synchronisation.

Chapter 6 concludes the thesis with a summary of the results of the investigation into the synthesis and axiomatisation problems for the structural equivalences, isomorphism and duplication equivalence, and describes possible directions for future work, including the extension of the synthesis and axiomatisation problems to behavioural equivalences.

Appendix A provides a list of cross references for the main concepts and definitions, and Appendix B gives a summary of the different subsets of the Petri Box Calculus that are considered during the various investigations into the synthesis and axiomatisation problems.

Chapter 2

Properties

2.1 Introduction

This chapter begins by describing the alternative top-down and bottom-up approaches to solving the synthesis problem. In Section 2.2.3, the motivation for the selection of one of these approaches for further investigation is given. In Section 2.4, the relationship between the synthesis and axiomatisation problems is investigated. A collection of problems related to the synthesis and axiomatisation problems are presented, and the relationships between the time complexity of these problems are discussed.

In the second half of the chapter, various definitions and properties of Petri nets are described. Some definitions provide further insight into the relationship between box expressions, and the structure of members of the class of Petri boxes, while others will be used later for the analysis of nets given as input to the synthesis algorithm. In that sense, the definitions and properties described in this chapter can be thought of as a collection of tools for use in synthesising expressions from nets.

2.2 Solving the synthesis problem

In this section the alternative approaches of top-down synthesis and bottom-up synthesis are described, and their advantages and disadvantages discussed. In order to decide which of the two methods is most worthy of further investigation it is perhaps important to consider the properties that a good solution to the synthesis problem should have.

Useful criteria for selecting the approach to investigate include the ability of the method to produce a solution on any input, the efficiency of the method, and the relationship between the method and the problem of producing an axiomatisation. The ability to find an axiomatisation relies on analysing the points of non-determinism in producing a synthesised expression. Where there is a choice in the synthesis process allowing several syntactically different but semantically equivalent expressions to be generated, any axiomatisation must be able to equate the different forms of expressions that can be produced. The relationship between the synthesis problem and generating an axiomatisation is investigated further in Section 2.4.

If the synthesis problem is NP hard, then there is little hope of finding an algorithm that is both efficient and guarantees to find a solution on every input. It is also less likely that a heuristic approach will provide enough insight into the problem to allow an axiomatisation to be generated. In this respect, the importance of an efficient solution should be regarded as secondary to the other criteria.

In describing the top-down and bottom-up approaches to synthesis, the simple net in Figure 2.1 is considered as an example input.

2.2.1 Top-down approach

The top-down approach to synthesis requires that the main connective of the synthesised expression can be found by analysing the structure of the input net. The input net can then be decomposed into a collection of smaller nets

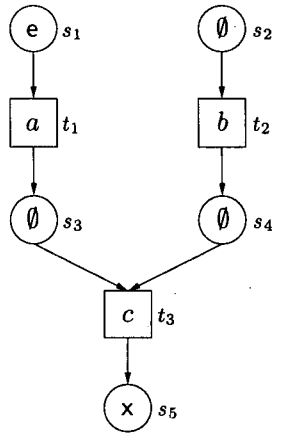


Figure 2.1: Implementation of $E = (a \parallel b); c$

corresponding to the subexpressions of the synthesised expression. The process is applied recursively to each of the decomposed subnets until no further decomposition can be applied. Each step of the recursion adds further detail to the expression being synthesised, finally resulting in an expression whose implementation is equivalent to the input net.

For example, consider the net in Figure 2.1, and the problem of synthesising an expression whose implementation is isomorphic to that net. Using the top-down approach, the synthesised expression would initially be set to E, where E is the unknown expression corresponding to the input net. An analysis of the structure of the net determines that the main connective of E is the sequence operator, and that the input net can be decomposed into two components. Therefore, the synthesised expression can be refined from E to $E_1; E_2$ where E_1 and E_2 are unknown expressions corresponding to the two decomposed subnets. A recursive application of the top-down synthesis process to the first subnet determines that the main connective of E_1 is the parallel composition operator and that this subnet can be decomposed into two further components. At this stage, the synthesised expression is $(E_3 \parallel E_4); E_2$ where E_2, E_3 and E_4 are unknown expressions corresponding respectively to implementations of atomic actions with labels c, a and b. Hence, after further analysis, the

recursion terminates with the output expression $(a \parallel b); c$.

The top-down approach takes a global view, and tries to decompose the problem into a collection of smaller problems. From the point of view of synthesis, the size of the problem corresponds with the size of the input net. For the top-down approach to be successful, there needs to be a strong relationship between the structure of the net and the structure of the expression from which the net was derived. While this relationship certainly seems to exist for operators such as parallel, choice, sequence and iteration, it is not so clear for operators such as synchronisation and restriction. For example, when the restriction operator is applied to a net, the structure of the net may alter radically when transitions are removed.

In the description above, the synthesis process proceeds directly to a solution – *i.e.* once a particular connective has been identified for the synthesised expression, it is not necessary to revise the choice of connective in the light of new information. Should a situation arise, where the choice of connectives for the synthesised expression is found to be incorrect, then the synthesis algorithm would need to backtrack to the point where the wrong choice was made, and a different search path taken. Any algorithm in which backtracking is necessary is likely to be both less efficient and less amenable to the production of an axiomatisation than an algorithm which does not involve backtracking.

2.2.2 Bottom-up approach

The bottom-up approach to synthesis begins by synthesising expressions for small pieces of the input net, then performs an analysis of the interfaces between the pieces of net corresponding to the already synthesised subexpressions to determine how those subexpressions should be combined.

For example, for the net in Figure 2.1, three subexpressions, $E_1 = a$, $E_2 = b$ and $E_3 = c$ are synthesised from the transitions, t_1, t_2 and t_3 respectively. An analysis of the pre and post places of these transitions determines that t_1 and t_2 share no common places, and there is a directed path from t_1 and t_2

to t_3 via places s_3 and s_4 respectively. The analysis allows the conclusion that t_1 and t_2 were combined using parallel composition, and the resulting net composed in sequence with an atomic action t_3. Therefore, the three synthesised subexpressions are combined to give $(E_1 \parallel E_2); E_3$, resulting in the final expression $(a \parallel b); c$.

The bottom-up approach only requires a local analysis of the input net, and proceeds by analysing the relationships between sets of transitions which share common places. This approach is certainly more suitable than the top-down method for providing a partial solution when the input net is not exactly equivalent to the implementation of a box expression. It also seems that the bottom-up approach may be better suited to dealing with input nets that involve the synchronisation, and particularly the restriction operator.

The interfaces between subnets for which expressions have already been synthesised are easily identified, since they consist of those places that are common to the subnets. For example, in Figure 2.1, s_3 and s_4 are the common places between the subnet containing t_1 and t_2 and the subnet containing t_3. However, there is likely to be many interfaces to analyse at each step of the synthesis process, and it may be difficult to determine a suitable order to deal with the interfaces. This problem may lead to the use of heuristics and the possibility of backtracking in a solution to the synthesis problem, making the problem of generating an axiomatisation much more difficult.

2.2.3 Choice of method

For both approaches to the synthesis problem, there is some uncertainty that the method will provide a good solution, partly because it is not yet known how inherently difficult the synthesis problem is. The main motivation behind the investigation into the synthesis problem is to provide a framework for generating an axiomatisation of the Petri Box Calculus. The top-down approach appears to give a better framework for the production of an axiomatisation, provided that the main connective of the synthesised expression can be found

by analysing the structure of the input net, and the synthesis process can proceed without resorting to heuristics or backtracking. For this reason, the top-down approach to synthesis is chosen for further investigation.

The compositional nature of the semantics for the parallel, sequence, choice and iteration operators suggest that a top-down decomposition should be possible for input nets that have been constructed using only these operators. However, it may not be so easy to synthesise expressions from nets that involve the synchronisation and restriction operators. Therefore, an initial investigation into a subset of the Petri Box Calculus that does not contain these operators is more likely to produce results than immediately attempting to deal with **sy** and **rs** .

2.3 Example

In this section the synthesis of an example net, using a top-down approach, is described. The algorithm presented here is limited in that it is only applicable to a very small subset of the Petri Box Calculus, whose syntax is given in Table 2.1. The synthesis algorithm of this section is predicated on the equivalence relation of isomorphism. The purpose of presenting the algorithm is to introduce a general framework for synthesis, and to illustrate the type of problems that need to be solved when extending the scope of synthesis to larger subsets of the box calculus.

$E ::=$	α	Atomic action
	$\mid E \parallel E$	Parallel composition
	$\mid E; E$	Sequential composition

Table 2.1: Small subset of the box expression syntax

2.3.1 Algorithm

The synthesis algorithm takes as input a net, Σ, which is an implementation of a box expression from the syntax in Table 2.1. The output of the algorithm is an expression, E, such that any implementation of E is isomorphic to the input net, Σ. The pseudo-code for an algorithm which synthesises a box expression from the input net using a recursive top-down approach is shown below.

```
BOX EXPRESSION SYNTHESIS(Σ)
1   type=ANALYSE(Σ)
2   if type=atomic then
3     return λ(t)
4   else
5     Σ1,Σ2=DECOMPOSE(Σ,type)
6     E1=BOX EXPRESSION SYNTHESIS(Σ1)
7     E2=BOX EXPRESSION SYNTHESIS(Σ2)
8     if type=parallel then
9       return E1 || E2
10    else
11      return E1; E2
12    end if
13  end if
```

The algorithm is based on the idea of a set of synthesis rules, with one rule for each operator in the box expression syntax - *i.e.* atomic actions, parallel composition, and sequence composition. The function ANALYSE determines which synthesis rule to apply, by examining structural properties of the input net. Using the syntax in Table 2.1, only one of the three synthesis rules will be applicable at each step of the algorithm. In general, when a larger subset of the Petri Box Calculus is used, there may be a choice of several synthesis rules to apply at each step. The pseudo-code for ANALYSE is presented below.

A formal definition for "Σ is disjoint" is deferred until Section 2.5.2, and the correctness of the decision procedure presented here is a result of Propositions 3 and 5 in Chapter 3.

```
ANALYSE(Σ = (S, T, W, λ))
1   if |T| = 1 then
2     return atomic
3   else
4     if Σ is disjoint then
5       return parallel
6     else
7       return sequence
8     end if
9   end if
```

2.3.2 Synthesis rules

The synthesis rule for atomic actions forms the base case of the recursion. When Σ is isomorphic to the implementation of an atomic action, then $E = \alpha$, where α is the label of the single transition in Σ is a solution. The synthesis rules for parallel and sequence composition decompose Σ into a pair of smaller nets, Σ_1 and Σ_2. The output for the synthesis algorithm is obtained by recursively synthesising expressions E_1 and E_2 using Σ_1 and Σ_2 as input nets. E_1 and E_2 are combined as $E_1 \parallel E_2$ $(E_1; E_2)$ when the synthesis rule being applied is parallel (sequence).

The DECOMPOSE function, which performs the decomposition is not described explicitly in this section. A simple decomposition scheme which is sufficient to deal with the example input net in Figure 2.2 is:

Parallel composition: The net is decomposed into two connected components.

Sequential composition: The set of internal places in the input net which corresponds to the interface created by the semantics for sequential composition is identified. The input net is decomposed into two disjoint subnets by analysing the arcs from and to the set of internal places, and reconstructing the original interfaces.

This scheme does not cope with nets obtained from arbitrary expressions from the syntax in Table 2.1. For example, an implementation of $(a \parallel b) \parallel c$ contains three connected components. The net is disjoint, so the parallel composition synthesis rule is applied. However, the decomposition scheme requires that the net is decomposed into two connected components, which is not possible. A generally applicable decomposition scheme is described in detail in Chapter 3.

2.3.3 Example execution of the algorithm

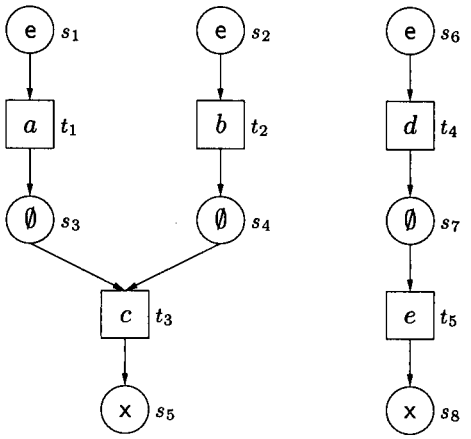


Figure 2.2: Example input to the synthesis algorithm

The net in Figure 2.2 is constructed from the box expression:

$$E = ((a \parallel b); c) \parallel (d; e)$$

Table 2.2 and Figure 2.3 describe the execution of Box Expression Synthesis, when given the net in Figure 2.2 as input. Each line in Table 2.2 corresponds to an execution of the body of Box Expression Synthesis.

The depth of recursion of each step of execution is shown in the *Depth* column. The depth of recursion corresponds to the distance from the root of the tree in Figure 2.3. The input net to the particular execution of BOX EXPRESSION SYNTHESIS is named in the *Net* column, together with the name of the expression to be synthesised in the *Exp* column. For the purposes of this example, the expression E_i corresponds to the net Σ_i, for all values of i. Each of the nets involved in the execution of the synthesis algorithm are contained in Figure 2.3. The synthesis rule to be applied is shown in the *Rule* column of Table 2.2. If the rule to be applied is sequence or parallel, then the names of the pair of nets obtained by decomposing the net are given in successive columns. The final two columns of Table 2.2 show how the expression from the *Exp* column is refined by the application of the synthesis rule, and how that refinement affects the overall synthesised expression. The path of execution in Figure 2.3 proceeds in a depth first fashion, and from left to right.

Step	Depth	Net	Exp.	Rule	Subnet 1	Subnet 2	Expression	Synthesis
1	0	Σ	E	parallel	Σ_1	Σ_2	$E_1 \parallel E_2$	$E_1 \parallel E_2$
2	1	Σ_1	E_1	sequence	Σ_3	Σ_4	$E_3; E_4$	$(E_3; E_4) \parallel E_2$
3	2	Σ_3	E_3	parallel	Σ_5	Σ_6	$E_5 \parallel E_6$	$((E_5 \parallel E_6); E_4) \parallel E_2$
4	3	Σ_5	E_5	atomic	--	--	a	$((a \parallel E_6); E_4) \parallel E_2$
5	3	Σ_6	E_6	atomic	--	--	b	$((a \parallel b); E_4) \parallel E_2$
6	2	Σ_4	E_4	atomic	--	--	c	$((a \parallel b); c) \parallel E_2$
7	1	Σ_2	E_2	sequence	Σ_7	Σ_8	$E_7; E_8$	$((a \parallel b); c) \parallel (E_7; E_8)$
8	2	Σ_7	E_7	atomic	--	--	d	$((a \parallel b); c) \parallel (d; E_8)$
9	2	Σ_8	E_8	atomic	--	--	e	$((a \parallel b); c) \parallel (d; e)$

Table 2.2: Example execution of the synthesis algorithm

In the following, the execution of BOX EXPRESSION SYNTHESIS(Σ) is described, where Σ is the net in Figure 2.2, and at the root of the tree in Figure 2.3. Each step of the execution given in Table 2.2 is described in more detail:

Step 1: The call to ANALYSE(Σ) determines that the input net is disjoint, and contains more than one transition. Therefore, ANALYSE returns

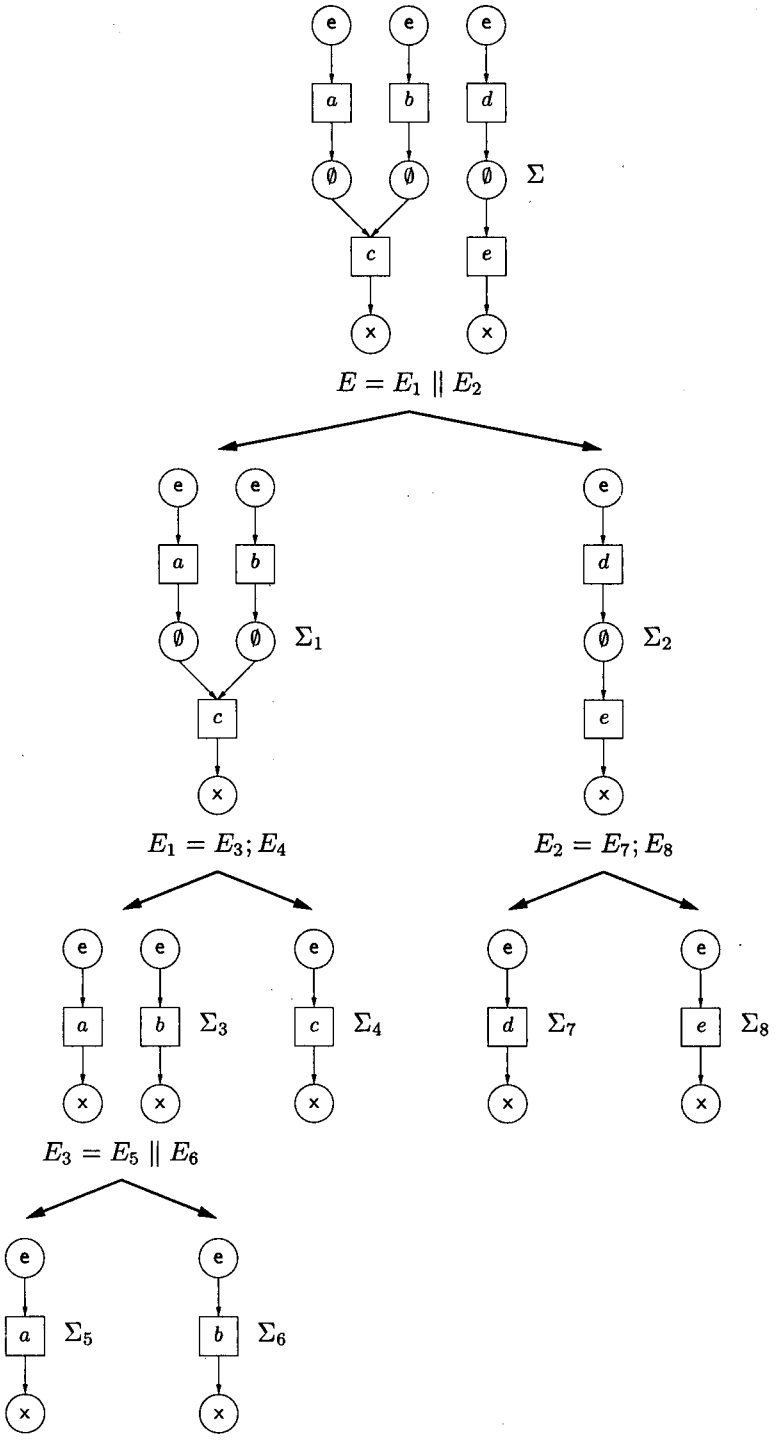


Figure 2.3: Example execution of the synthesis algorithm

parallel as the synthesis rule to be applied. Σ is decomposed into two connected components Σ_1 and Σ_2, shown in Figure 2.3, and the synthesised expression is refined to be $E = E_1 \parallel E_2$. Recursive calls to BOX EXPRESSION SYNTHESIS are made to synthesise expressions for Σ_1 and Σ_2.

Step 2: BOX EXPRESSION SYNTHESIS(Σ_1) is called from Step 1 to recursively synthesise an expression for Σ_1, and to refine E_1 in $E = E_1 \parallel E_2$. The analysis of Σ_1 determines that the sequence synthesis rule should be applied. The pair of internal places in Σ_1 is decomposed, and the nets, Σ_3 and Σ_4 are obtained. Hence, E_1 and E are refined to $E_1 = E_3; E_4$, and $E = (E_3; E_4) \parallel E_2$ respectively. Finally, the recursive calls BOX EXPRESSION SYNTHESIS(Σ_3) and BOX EXPRESSION SYNTHESIS(Σ_4) are made.

Step 3: The last step recursively calls BOX EXPRESSION SYNTHESIS(Σ_3) to find the expression for E_3. The parallel synthesis rule is applied, since Σ_3 is disjoint. Σ_3 is decomposed into Σ_5 and Σ_6, refining E_3 to $E_3 = E_5 \parallel E_6$. Hence E is refined to $E = ((E_5; E_6); E_4) \parallel E_2$. A further level of recursion is performed to find the expressions E_5 and E_6.

Step 4: The net Σ_5 is analysed, and found to contain only a single transition. Hence the atomic action synthesis rule is applied, and no further decomposition of the net takes place. The expression E_5 becomes fully refined (*i.e.* there are no unknown subexpressions) to $E_5 = a$. Hence, at the end of this step, the synthesised expression has been refined to $E = ((a; E_6); E_4) \parallel E_2$. Step 4 does not make any recursive calls to BOX EXPRESSION SYNTHESIS.

Step 5: BOX EXPRESSION SYNTHESIS(Σ_6) is called from Step 3, and proceeds in a manner similar to Step 4, refining E_6 to $E_6 = b$ and hence E to $E = ((a; b); E_4) \parallel E_2$.

Step 6: The second recursive call, BOX EXPRESSION SYNTHESIS(Σ_4), made from Step 2, applies the atomic action synthesis rule to refine E_4 to $E_4 = c$. Hence, E becomes $E = ((a;b);c) \parallel E_2$, and the control of execution returns to Step 1.

Step 7: Step 1 makes a second recursive call, this time to synthesise an expression for the net Σ_2. The analysis of Σ_2 determines that the sequence synthesis rule should be applied, and the net is decomposed into Σ_7 and Σ_8. The expression, E_2, is refined to $E_2 = E_7; E_8$, refining the output expression, E, to $E = ((a;b);c) \parallel (E_7; E_8)$. Finally, two recursive calls, to synthesise expressions for Σ_7 and Σ_8 are made.

Step 8: Σ_7 contains a single transition. Therefore, the atomic action synthesis rule is applied, and E_7 is refined to $E_7 = d$. Hence, E becomes $E = ((a;b);c) \parallel (d; E_8)$. No recursive calls to BOX EXPRESSION SYNTHESIS are made during this step of the execution.

Step 9: The second recursive call made in Step 7 is BOX EXPRESSION SYNTHESIS(Σ_8). Again, the atomic action synthesis rule is found to be applicable, and E_8 is refined to $E_8 = e$, producing the fully refined output expression $E = ((a;b);c) \parallel (d;e)$. Once the expression is fully refined, there are no outstanding recursive calls to be dealt with, and the execution can terminate.

2.3.4 Discussion

The algorithm for BOX EXPRESSION SYNTHESIS, described in Section 2.3.1 directly refines the output expression during the synthesis process. In practice, it would be more beneficial to construct a parse tree representation of the expression, especially if the expression is to be manipulated in some way once it has been synthesised. Figure 2.4 shows the parse tree for the expression $E = ((a;b);c) \parallel (d;e)$ synthesised from the input net given in Figure 2.2. There

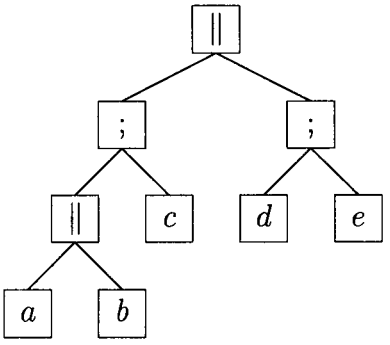


Figure 2.4: Parse tree of the synthesised expression

is a strong correspondence between the parse tree and the manner in which the recursive calls to BOX EXPRESSION SYNTHESIS are made, as illustrated in Figure 2.3.

The algorithm described in this section presents a general framework for the solution of the synthesis problem when the notion of equivalence being used is isomorphism. In Chapter 5, the extension of the framework to support synthesis under weaker structural equivalences is described, and Chapter 6 contains a discussion of how the synthesis problem may be approached when behavioural equivalences are being used. In the remainder of this section, the approach to solving the synthesis problem, using isomorphism as the equivalence relation, is discussed.

Given a subset of the Petri Box Calculus for which a synthesis algorithm is to be designed, there are two main areas for investigation, relating to different aspects of the synthesis rules. Firstly, a decision procedure needs to be developed to identify which of the synthesis rules can be applied at each step of the synthesis process. Secondly, for each synthesis rule, a method needs to be produced which will decompose the input net into a collection of smaller nets, while simultaneously refining the output expression. Normally, there will be a synthesis rule for each operator in the subset of the box calculus for which the synthesis algorithm is being developed.

To produce a decision procedure to determine which synthesis rule is to

be applied, the characteristic structural properties of the implementations of each type of box expression need to be investigated. For each synthesis rule, which corresponds to a particular type of expression, what is required is a property that holds for every net obtained from an expression of that type, and does not hold for any net that cannot be obtained from an expression of that type. Then, whenever that particular property is found to hold for the input net, it will be known that the synthesis rule is applicable. There may be several different, but equally suitable structural properties that can be used. For example, for the fragment of the box calculus in Table 2.1, either of the following properties could be used to identify that the atomic action synthesis rule should be applied:

- The net contains exactly one transition.
- The net contains exactly two places.

Some operators in the Petri Box Calculus are redundant. For example, any expression involving the scoping operator has an equivalent form which uses synchronisation and restriction in place of scoping. There is no need to have a synthesis rule for redundant operators. In other cases, there may be a partial overlap between operators. For example, the choice expression $(a \parallel \widehat{a}) \,\square\, \emptyset$ has an implementation that is isomorphic to the synchronisation expression $(a \parallel \widehat{a})$ **sy** a. In such cases, several synthesis rules may be found to be applicable, and it does not matter which one is applied.

For each synthesis rule, the input net must be decomposed into smaller nets. This involves identifying the interfaces between components of the input net, and using them to reconstruct the original interfaces. When the synthesis rule corresponds to an associative operator, there may be several different interfaces that can be decomposed. The example input given in Figure 2.2 was chosen to avoid this problem arising. For example, the expressions $a; (b; c)$ and $(a; b); c$ have isomorphic implementations. When the sequence synthesis rule is applied, two interfaces will be identified, and the choice of the one to

decompose will affect the bracketing of the synthesised expression.

The parallel and sequence synthesis rules presented in this section decompose the input net into two smaller nets, Σ_1 and Σ_2. For the sequence synthesis rule, it is possible to order the pair of nets obtained from the decomposition of the input net, by setting Σ_1 (Σ_2) to be the net that contains the entry (exit) places of the input net. However, for the parallel synthesis rule, there are two possibilities for the assignment to Σ_1 and Σ_2. For example, in Step 1 of the execution given in Table 2.2, the assignment to Σ_1 and Σ_2 (shown in Figure 2.3) could have been reversed. This would result in the output expression produced by BOX EXPRESSION SYNTHESIS being $E = (d; e) \parallel ((a; b); c)$. The implementation of E is isomorphic to the input net shown in Figure 2.2. The possibility of this type of alternative decomposition arises when the synthesis rule corresponds to an operator that is commutative (such as parallel composition). A scheme for dealing with the problems caused by the associativity and commutativity properties of operators in the Petri Box Calculus is presented as part of the investigation carried out in Chapter 3.

2.4 Relationship between synthesis and axiomatisation problems

In this section, the way in which an analysis of a synthesis algorithm can be used to produce an axiomatisation is described. As with the previous section, the equivalence relation used is isomorphism, and it is left until Chapter 5 to explain how the technique can be extended to other structural equivalences, such as duplication equivalence. In Chapter 6, there is a discussion on how the problem of finding an axiomatisation for behavioural equivalences may be tackled.

Corresponding to the framework for the synthesis algorithm described in Section 2.3.1, there is a general framework for a formal verification that the

algorithm is correct. It is this framework that can be used to guide the production of an axiomatisation, and the verification of the synthesis algorithm also serves as a proof that the axiom system obtained is complete. The remainder of this section describes the framework for the verification of the synthesis algorithm, and the techniques by which sets of axioms can be found.

2.4.1 Verification of the synthesis algorithm

The framework for the synthesis algorithm consists of a set of synthesis rules, and a corresponding set of properties used to identify which synthesis rule to apply. In general, each synthesis rule contains a method for the decomposition of the input net into smaller nets, and a means for representing the decomposition using a box expression. In verifying that the synthesis algorithm is correct, the following properties need to be shown:

- For any net that is isomorphic to the implementation of some box expression, the synthesis algorithm will synthesise an expression from that net.

- For any execution of the synthesis algorithm the implementation of the output expression will be isomorphic to the input net.

The proof that the synthesis algorithm is correct relies on showing that the decision procedure used to select the synthesis rule to apply works, and that the decomposition scheme for each synthesis rule is sound. In addition, several support proofs are needed to tie all the results together. The support proofs will be much the same for synthesis algorithms for different subsets of the Petri Box Calculus given in Table 1.1.

The decision procedure used to identify the synthesis rule to apply effectively associates a set of preconditions with each synthesis rule. The preconditions for a synthesis rule must hold before the rule can be applied. For structural equivalences, such as isomorphism, the preconditions take the form

of structural properties of the input net. For each synthesis rule, a proof is required that shows that whenever the preconditions hold, then there exists an expression whose main connective is the same as that of the refined expression produced by the synthesis rule, and the implementation of that expression is isomorphic to the input net.

The second proof associated with each synthesis rule is required to show that the decomposition performed by the synthesis rule is sound. The synthesis rule decomposes the input net into a collection of smaller subnets. The proof must show that each decomposed subnet is isomorphic to the implementation of a box expression, and that when the subnets are recombined according to the refinement made to the synthesised expression by the synthesis rule, then a net isomorphic to the input net is obtained.

Two support proofs complete the verification of correctness for the synthesis algorithm. The first shows that for any net isomorphic to the implementation of a box expression, at least one of the synthesis rules is applicable to that net. The second puts all of the results together, using an inductive argument, to show that on any valid input, the synthesis algorithm will produce the correct output.

2.4.2 Obtaining an axiom system

There are two (possibly empty) sets of axioms associated with each synthesis rule. The first set of axioms is related to the structural properties used as preconditions for the synthesis rule, and are such that for any expression whose implementation is a net that satisfies the preconditions, the axioms can be used to rewrite that expression into the form of the refined expression produced by applying the synthesis rule.

The second set of axioms is associated with the decomposition method of the synthesis rule, and is affected by the amount of non-determinism present in the method for decomposition. For each synthesis rule, all possible ways of decomposing the input net and/or refining the synthesised expression must

be analysed. By the soundness proof for net decomposition, the different outcomes of a non-deterministic decomposition method must be equivalent – either immediately, or after further applications of synthesis rules. For the purposes of analysis, and producing an axiomatisation, it is better if the decomposition method can be designed so all outcomes are equivalent after a single application of a synthesis rule. The set of axioms must be such that it is possible to rewrite between all possible outcomes introduced by the non-determinism in the synthesis rule.

If all of the synthesis rules can be made to produce completely deterministic results, then the expression obtained from the synthesis process will be in a canonical form. However, it is not imperative to define a canonical form, or eliminate every point of non-determinism. The cost is the extra analysis required to produce an axiomatisation. The benefit of this flexibility is that it may be easier to design an efficient synthesis algorithm. In practice there will be a balance between the amount of work that is done in eliminating non-determinism from the synthesis algorithm, and the amount of work done in finding the set of axioms associated with the decomposition scheme for each synthesis rule.

2.4.3 Related problems

In this section, several problems that are related to the synthesis algorithm are introduced. Each of these problems extend the scope of BOX EXPRESSION SYNTHESIS in some way. The extensions described here include the problem of synthesising a unique, or canonical, expression from the input net, and the problem of checking whether two Petri boxes are equivalent. These two problems are recast in purely algebraic terms – *i.e.* the algorithm which solves the problem must work entirely in the domain of box expressions. This is an important consideration from the point of view of efficiency, because implementations of certain forms of box expression are exponential in the size of the expression. Finally, the automatic generation of a proof of equivalence for two

box expressions is considered. All of the problems described in this section are introduced in terms of the net equivalence isomorphism. The majority of the problems have analogues for other notions of equivalence, such as duplication equivalence. As well as describing each of the extensions, the complexity theoretic relationships between BOX EXPRESSION SYNTHESIS and the problems introduced in this section are investigated.

Domain of Petri boxes

CANONICAL BOX EXPRESSION SYNTHESIS imposes the extra condition on the synthesis process that the expression produced by the algorithm is in canonical form. This means that given two isomorphic nets, Σ_1 and Σ_2, exactly the same expression is produced by CANONICAL BOX EXPRESSION SYNTHESIS(Σ_1), and CANONICAL BOX EXPRESSION SYNTHESIS(Σ_2). This property is not true for BOX EXPRESSION SYNTHESIS, which may contain elements of non-determinism. The extension from the standard synthesis algorithm to one which produces canonical form expressions essentially involves the elimination of all points of non-determinism in the synthesis process. The non-determinism of the decomposition performed by each synthesis rule can be eliminated independently of the rest of the algorithm. If all of the synthesis rules are completely deterministic, then a canonical expression will be synthesised. Alternatively, an expression can be synthesised as normal, and then manipulated into a canonical form. This approach involves the explicit definition of the form of canonical expressions.

CANONICAL BOX EXPRESSION SYNTHESIS

INSTANCE: Net, Σ, member of the class of Petri boxes.

SOLUTION: Canonical box expression, E, such that $\mathsf{box}(E) = [\Sigma]$.

PETRI BOX ISOMORPHISM

INSTANCE: Nets, Σ_1, Σ_2, members of the class of Petri boxes.

QUESTION: Is $\Sigma_1 =_{iso} \Sigma_2$?

The problem of PETRI BOX ISOMORPHISM is to check whether two nets, which are implementations of box expressions, are isomorphic to each other. It is easy to see that given a solution to CANONICAL BOX EXPRESSION, an algorithm for PETRI BOX ISOMORPHISM can be constructed as follows:

```
PETRI BOX ISOMORPHISM(Σ1, Σ2)
1   E1=CANONICAL BOX EXPRESSION SYNTHESIS (Σ1)
2   E2=CANONICAL BOX EXPRESSION SYNTHESIS (Σ2)
3   if E1 = E2 then
4     return yes
5   else
6     return no
7   end if
```

PETRI BOX ISOMORPHISM is a restricted case of the more general graph isomorphism problem. The graph isomorphism problem is interesting because it is one of very few well known problems for which the complexity of the problem has not been settled. No proof showing that the graph isomorphism problem is NP-complete has been produced, nor has a polynomial time algorithm been developed. There are classes of graphs for which polynomial time algorithms are known. For example isomorphism of trees, and planar graphs can be checked in polynomial time. However, the class of Petri boxes does not seem to be a subset of any of the classes of graphs for which polynomial time algorithms are known. It has been shown that an efficient solution to CANONICAL BOX EXPRESSION SYNTHESIS provides an efficient method for checking the isomorphism of Petri boxes. Hence, the investigation into the synthesis problem may provide some insight into the graph isomorphism problem. In Chapter 4, it is shown that the Petri Box Calculus of Table 1.1 is expressive enough to encode arbitrary instances of the graph isomorphism problem. The implication of this result is that for sufficiently large subsets of the Petri

Box Calculus, the problem of extending the synthesis algorithm to produce canonical form expressions has the same complexity as the graph isomorphism problem.

The close relationship between synthesising canonical form expressions, and the graph isomorphism problem motivates an investigation into graph isomorphism. The following describes how an efficient solution to the graph isomorphism problem allows BOX EXPRESSION SYNTHESIS, for any subset of the Petri Box Calculus, to be extended to CANONICAL BOX EXPRESSION SYNTHESIS. Although the complexity of the graph isomorphism problem is not known, there are algorithms based on a heuristic approach that perform well in practice [40].

GRAPH ISOMORPHISM
INSTANCE: Graphs $G = (V, E)$, $G' = (V, E')$
QUESTION: Are G and G' "isomorphic", that is, is there a one-to-one function $f : V \to V$ such that $\{u, v\} \in E$ if and only if $\{f(u), f(v)\} \in E'$?

Equivalent in complexity to GRAPH ISOMORPHISM is the problem of finding a canonical labelling ([40]) for the nodes of the graph. A method for obtaining such a labelling is not presented in this section. However, it is worth noting that tools such as *nauty* [40] provide the facility for canonically relabelling a given graph. The following pseudo-code shows how CANONICAL BOX EXPRESSION SYNTHESIS can be implemented in terms of BOX EXPRESSION SYNTHESIS, and CANONICAL RELABEL, a routine that returns a canonically labelled graph isomorphic to the given graph.

```
CANONICAL BOX EXPRESSION SYNTHESIS(Σ)
1  Σ'=CANONICAL RELABEL(Σ)
2  return BOX EXPRESSION SYNTHESIS(Σ')
```

Any non-determinism in the BOX EXPRESSION SYNTHESIS is due to the different ways in which isomorphic nets can be represented (*i.e.* different la-

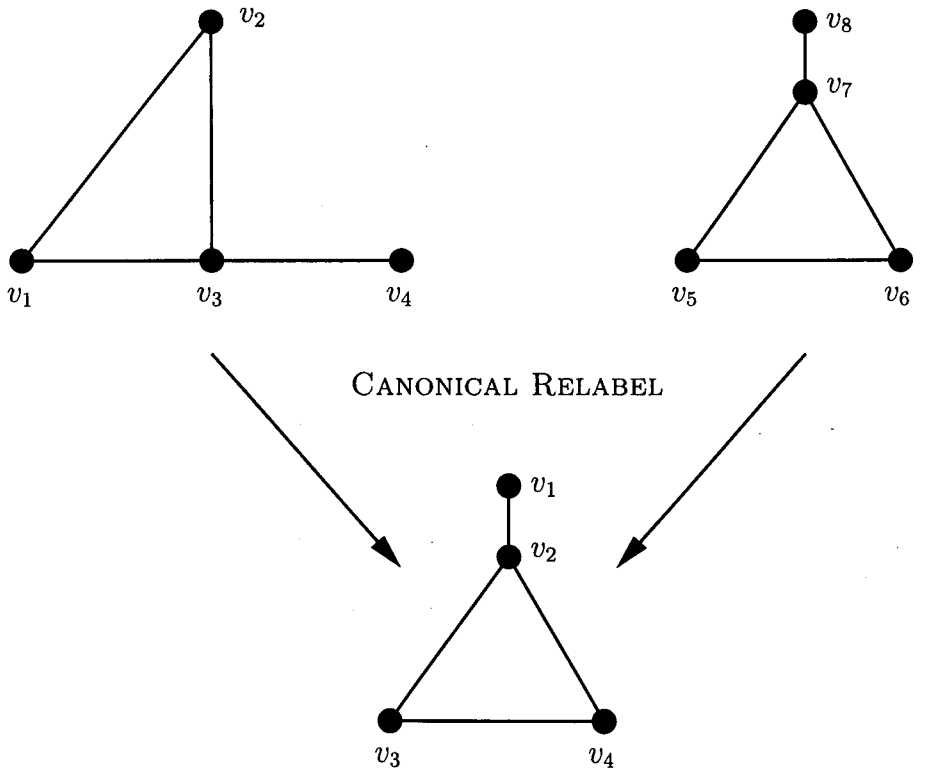


Figure 2.5: Canonical relabelling of a graph

bellings for the node names in the net). The call to CANONICAL RELABEL ensures that isomorphic graphs (or nets) have identical node name labellings – Figure 2.5 illustrates the behaviour of CANONICAL RELABEL on a pair of isomorphic graphs. Hence, the synthesis algorithm produces a unique expression for each class of isomorphic nets. The downside of this method for producing canonical form expressions is that in order to obtain a complete axiomatisation, a detailed analysis of the points of non-determinism in the synthesis algorithm is still required.

Domain of box expressions

A solution to CANONICAL BOX EXPRESSION SYNTHESIS allows the canonical form for a box expression to be found. This can be achieved by constructing an implementation of the expression, then synthesising the canonical form

expression from the net. Similarly, a solution to PETRI BOX ISOMORPHISM can be used to check whether two expressions are equivalent. The problem with this approach is that a net needs to be constructed from expressions, and in some cases the size of the net will be exponential in the size of the expression. For example, the implementations of the following expressions are exponential in size:

$$E = (a \parallel ... \parallel a) \,\square ... \square\, (a \parallel ... \parallel a)$$
$$F = [a * a * [a * a * [...[a * a * a]...]]]$$

An implementation of E contains an exponential number of entry and exit places in comparison to the number of actions in the expression. When nested iteration expressions are implemented in terms of nets, the net contains at least 2^n transitions for n levels of nesting.

These observations lead to the question whether the problems of finding the canonical form of an expression, and checking whether two expressions are equivalent, can be solved without resorting to the domain of Petri boxes.

CANONICAL BOX EXPRESSION

INSTANCE: Box expression, E

SOLUTION: Canonical box expression, E', such that $\mathsf{box}(E) = \mathsf{box}(E')$.

BOX EXPRESSION ISOMORPHISM

INSTANCE: Box expressions E_1, E_2.

QUESTION: Is $\mathsf{box}(E_1) = \mathsf{box}(E_2)$?

It is likely that the algorithms for CANONICAL BOX EXPRESSION and BOX EXPRESSION ISOMORPHISM will work by manipulating the parse trees for the input expressions. A solution to these problems can be derived from the corresponding problems in the domain of Petri boxes by analysing the way in which the synthesis rules for the algorithm CANONICAL BOX EXPRESSION SYNTHESIS refine the expression that is synthesised. This analysis should allow the algorithm to be abstracted to the level of box expressions. In general,

it seems that if there is a polynomial time algorithm for CANONICAL BOX EXPRESSION SYNTHESIS, then there will be corresponding polynomial time algorithms for CANONICAL BOX EXPRESSION and BOX EXPRESSION ISOMORPHISM. Even if an input net to the synthesis algorithm has exponential size, the synthesised expression, and therefore the number of refinements to the expression, will be small.

BOX EXPRESSION ISOMORPHISM PROOF
INSTANCE: Box expressions E_1, E_2, such that $\mathsf{box}(E_1) = \mathsf{box}(E_2)$.
SOLUTION: A proof that the expressions are equivalent.

A natural extension of BOX EXPRESSION ISOMORPHISM, showing that two expressions are equivalent, is BOX EXPRESSION ISOMORPHISM PROOF, which automatically generates a proof that the expressions are equivalent. A proof generated by BOX EXPRESSION ISOMORPHISM PROOF will take the form of a series of applications of axioms, such as that seen in Section 1.5. Recall that there are two sets of axioms associated with each synthesis rule. This means that up to two schemes for applying the axioms will be required for each synthesis rule used by the synthesis algorithm. As with CANONICAL BOX EXPRESSION and BOX EXPRESSION ISOMORPHISM, it should be possible to abstract away from the domain of Petri boxes, and work purely with box expressions. The algorithm for BOX EXPRESSION ISOMORPHISM PROOF will follow that of BOX EXPRESSION ISOMORPHISM, except that each manipulation that is carried out to the input expressions needs to be supported by an axiomatic proof that the manipulation is valid.

2.5 Definitions and properties

This section introduces the notation and definitions used by the synthesis algorithms algorithms presented in the following chapters. The first set of definitions is concerned purely with the domain of nets. These include the

classification of places and transitions in the net, the definition of connectedness properties in the net, the notion of clusters of places, and relations on the connectivity of transitions in the net. The remaining definitions have applications to both the domain of box expressions, and the domain of Petri boxes. An ordering over atomic actions (and hence transition labels) is defined. There is an investigation into the mapping between atomics actions in an expression, and the transitions in a net constructed from that expression. Finally, an auxiliary operator, $\odot$ is defined which is used to relate the synchronisation of actions in an expression, with the synchronisation of transitions in the corresponding net. For each of the following definitions, the motivation for the introduction of the definition, and its applications to the synthesis problem are discussed.

2.5.1 Classifying places and transitions

The following table defines various classifications for the nodes (*i.e.* places and transitions) of a labelled net, $\Sigma = (S, T, W, \lambda)$:

Name	Definition
Entry places	$S_{\mathsf{e}} = \{s \in S \mid \lambda(s) = \mathsf{e}\}$
Internal places	$S_{\mathsf{i}} = \{s \in S \mid \lambda(s) = \emptyset\}$
Exit places	$S_{\mathsf{x}} = \{s \in S \mid \lambda(s) = \mathsf{x}\}$
All nodes	$N_{\mathsf{a}} = S \cup T$
Internal nodes	$N_{\mathsf{i}} = S_{\mathsf{i}} \cup T$
Entry transitions	$T_{\mathsf{e}} = \{t \in T \mid \exists s \in S_{\mathsf{e}} : W(s,t) \neq 0\}$
Exit transitions	$T_{\mathsf{x}} = \{t \in T \mid \exists s \in S_{\mathsf{x}} : W(t,s) \neq 0\}$

The set of internal nodes contains only those nodes which do not form part of an entry or exit interface. The entry (exit) transitions are those transitions with an arc from an entry place (to an exit place). The notation $S_{\mathsf{e}}(\Sigma')$ is used to represent the entry places of the net Σ'. Similarly for the other classifications of nodes. When no net is specified, the net Σ should be assumed.

Recall, ${}^\bullet\Sigma$ and $\Sigma^\bullet$ are used to represent the set of entry places and exit places respectively. Hence ${}^\bullet\Sigma = S_{\mathsf{e}}$ and $\Sigma^\bullet = S_{\mathsf{x}}$. This notation is extended to single nodes, and sets of nodes. ${}^\bullet n$ and $n^\bullet$ are used to represent the set of pre and post nodes respectively, of a node, n. Similarly, ${}^\bullet N$ and $N^\bullet$ can be defined for a set of nodes, N.

$$\begin{aligned}
{}^\bullet n &= \{n' \in S \cup T \mid W(n', n) \neq 0\} \\
n^\bullet &= \{n' \in S \cup T \mid W(n, n') \neq 0\} \\
{}^\bullet N &= \{n \in S \cup T \mid \exists n' \in N : W(n, n') \neq 0\} \\
N^\bullet &= \{n \in S \cup T \mid \exists n' \in N : W(n', n) \neq 0\}
\end{aligned}$$

A place, s is isolated if it has no incoming or outgoing arcs. The set of all isolated places of a net, $\Sigma = (S, T, W, \lambda)$, is given by:

$$\mathcal{I}(\Sigma) = \{s \in S \mid \forall t \in T : W(s, t) + W(t, s) = 0\}$$

2.5.2 Connectedness properties

In this section, the relations $\overset{\leftrightarrow}{\sim}_N$ and $\overset{\rightarrow}{\sim}$ are defined. $\overset{\leftrightarrow}{\sim}_N$ is an undirected connectedness relation, defined over the domain N, some subset of N_{a}. $\overset{\rightarrow}{\sim}$ is a directed connectedness relation defined over the domain N_{a}. Figure 2.6 shows a net which is the parallel composition of two subnets, (i) and (ii). This net will be used to illustrate examples of these connectedness relations.

Undirected connectedness

For the undirected connectedness relation, $\overset{\leftrightarrow}{\sim}_N$, the direction of the arcs of the net are ignored. When the domain of the relation is the set of all nodes, the equivalence classes of $\overset{\leftrightarrow}{\sim}_{N_{\mathsf{a}}}$ correspond to the connected components of the net. The other domain of interest is the set of internal nodes, N_{i}. The undirected connectedness relation, $\overset{\leftrightarrow}{\sim}_N$, is the least relation satisfying the following

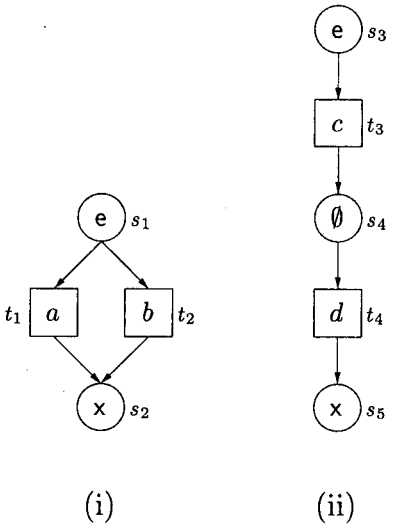


Figure 2.6: Example connectedness properties

properties:

$$\forall n \in N : n \overset{\leftrightarrow}{\sim}_N n$$
$$\forall n_1, n_2 \in N : W(n_1, n_2) \neq 0 \vee W(n_2, n_1) \neq 0 \Rightarrow n_1 \overset{\leftrightarrow}{\sim}_N n_2$$
$$\forall n_1, n_2, n_3 \in N : n_1 \overset{\leftrightarrow}{\sim}_N n_2 \wedge n_2 \overset{\leftrightarrow}{\sim}_N n_3 \Rightarrow n_1 \overset{\leftrightarrow}{\sim}_N n_3$$

The relation, $\overset{\leftrightarrow}{\sim}_N$, is an equivalence relation over the domain N. A net, Σ is *connected* if: $\forall n_1, n_2 \in N_{\mathsf{a}} : n_1 \overset{\leftrightarrow}{\sim}_{N_{\mathsf{a}}} n_2$, otherwise, Σ is *disjoint.* Σ is *internally connected* if: $\forall n_1, n_2 \in N_{\mathsf{i}} : n_1 \overset{\leftrightarrow}{\sim}_{N_{\mathsf{i}}} n_2$ otherwise Σ is *internally disjoint.*

Examples

The net in Figure 2.6 is disjoint because, for example $s_1 \overset{\leftrightarrow}{\not\sim}_{N_{\mathsf{a}}} s_3$. Both subnet (i) and subnet (ii) are connected. However, only subnet (ii) is internally connected. Subnet (i) is internally disjoint because $t_1 \overset{\leftrightarrow}{\not\sim}_{N_{\mathsf{i}}} t_2$.

Connected components

The undirected connectedness relation, $\overset{\leftrightarrow}{\sim}_{N_{\mathsf{a}}}$ can be used to define a mapping, $\mathcal{G} : 2^{S \cup T} \rightarrow 2^{S \cup T}$, which, for any set of nodes, N, gives the set of nodes in the

connected component(s) containing at least one node of N.

$$\mathcal{G}(N) = \{n \in N_{\mathsf{a}} \mid \exists n' \in N : n \overset{\leftrightarrow}{\sim}_{N_{\mathsf{a}}} n'\}$$

For example, in Figure 2.6: $\mathcal{G}(\{s_2, t_1\})$ returns the set of nodes in subnet (i), and $\mathcal{G}(\{s_4, s_1\})$ returns the set of all nodes.

The subcomponent of the net $\Sigma = (S, T, W, \lambda)$, that contains the set of nodes, $N \subseteq S \cup T$ is given by $\Sigma\,|_N$, where:

$$\Sigma\,|_N = (S \cap N, T \cap N, W\,|_{((S \cup T) \cap N) \times ((S \cup T) \cap N)}, \lambda\,|_{(S \cup T) \cap N})$$

Hence, for the net, Σ in Figure 2.6: $\Sigma\,|_{\mathcal{G}(\{s_2, t_1\})}$ is subnet (i), and $\Sigma\,|_{\mathcal{G}(\{s_4\})}$ is subnet (ii). Usually, N will be chosen to be some equivalence class of the relation, $\overset{\leftrightarrow}{\sim}_{N_{\mathsf{a}}}$, although the sequence synthesis rule of Chapter 3 uses $\Sigma\,|_N$ to extract a component from a connected net.

Directed connectedness

The directed connectedness relation, $\overset{\rightarrow}{\sim}$, is defined over the domain of the set of all nodes, N_{a}. The directed connectedness relation, $\overset{\rightarrow}{\sim}$, is the smallest relation satisfying the following properties:

$$\forall n_1, n_2 \in N_{\mathsf{a}} : W(n_1, n_2) \neq 0 \Rightarrow n_1 \overset{\rightarrow}{\sim} n_2$$
$$\forall n_1, n_2, n_3 \in N_{\mathsf{a}} : n_1 \overset{\rightarrow}{\sim} n_2 \wedge n_2 \overset{\rightarrow}{\sim} n_3 \Rightarrow n_1 \overset{\rightarrow}{\sim} n_3$$

For example, in Figure 2.6, $t_3 \overset{\rightarrow}{\sim} s_5$ and $s_1 \overset{\rightarrow}{\sim} s_2$, but, $s_2 \overset{\rightarrow}{\not\sim} s_1$ and $t_4 \overset{\rightarrow}{\not\sim} t_1$.

2.5.3 Clusters of places

An equivalence relation, $\simeq_p$, is defined on the set of places of a net, so that for any implementation, Σ, of a box expression, E, there is a correspondence between the equivalence classes of $\simeq_p$, and the applications of the $\otimes$ operator used in constructing an implementation of E. This correspondence is formalised by Proposition 6 in Section 3.4.

A binary relation, $\sim$, is defined on the places of a net, $\Sigma = (S, T, W, \lambda)$. A pair of places are related if they have a common pre- or post-transition – *i.e.*:

$$\forall s_1, s_2 \in S : s_1 \sim s_2 \Leftrightarrow \exists t \in T : W(s_1, t) \cdot W(s_2, t) + W(t, s_1) \cdot W(t, s_2) \neq 0$$

The relation $\sim$ is reflexive and symmetric. By taking the transitive closure of $\sim$, an equivalence relation, $\simeq_p$, is obtained. The equivalence classes of $\simeq_p$ partition the places of the net into *clusters*.

Figure 2.7 shows the net which is an implementation of the box expression:

$$(((a \parallel a) \,\square\, (a \parallel a)); a; (a \parallel a)) \parallel ((a \parallel a) \,\square\, a \,\square\, (a; a; a))$$

The relation, $\simeq_p$, partitions the places of this net into nine equivalence classes, p_1 to p_9, with, for example, $p_2 = \{s_5, s_6, s_7, s_8\}$.

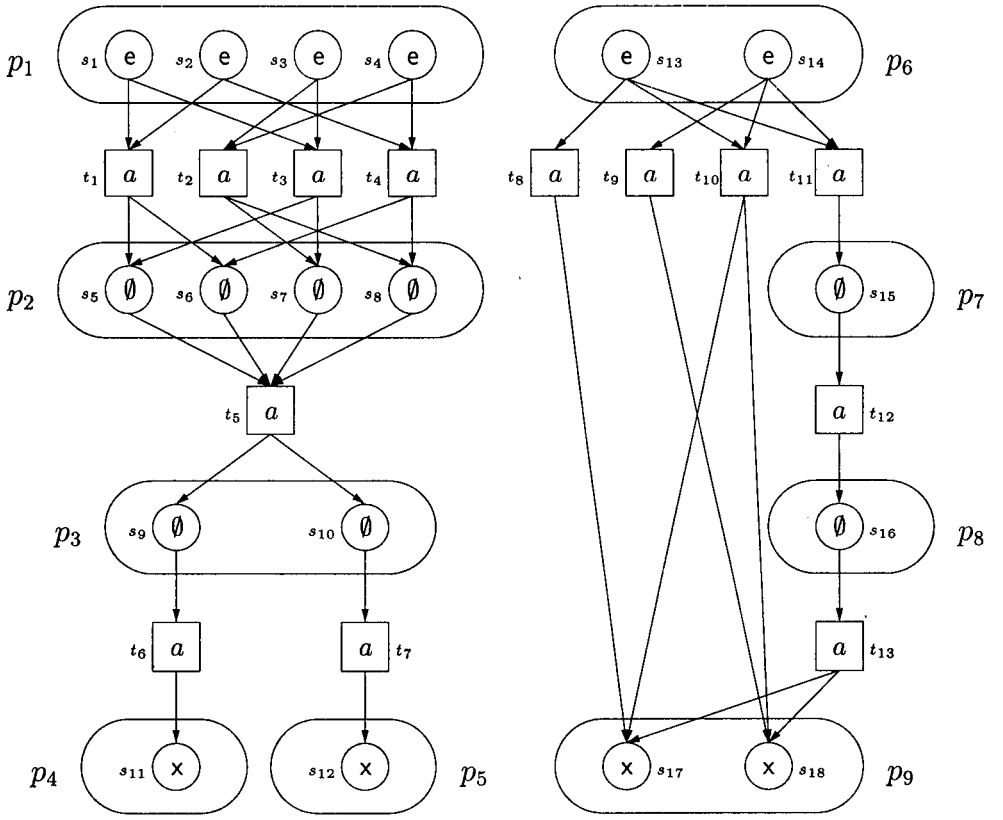


Figure 2.7: Partitioning the places into clusters

A function, $\mathcal{C} : S \rightarrow 2^S$ is used to obtain the equivalence class (cluster) of places to which a given place belongs:

$$\forall s \in S : \mathcal{C}(s) = \{s' \in S \mid s \simeq_p s'\}$$

The set of all clusters of internal places of a net, Σ, is given by:

$$\mathcal{C}_{\mathsf{i}}(\Sigma) = \{\mathcal{C}(s) \mid s \in S_{\mathsf{i}}\}$$

For example, for the net, Σ in Figure 2.7,

$$\begin{aligned} \mathcal{C}(s_3) &= \{s_1, s_2, s_3, s_4\} \\ \mathcal{C}_{\mathsf{i}}(\Sigma) &= \{\{s_5, s_6, s_7, s_8\}, \{s_9, s_{10}\}, \{s_{15}\}, \{s_{16}\}\} \end{aligned}$$

2.5.4 Connectivity of transitions

For a net $\Sigma = (S, T, W, \lambda)$, and transitions $t, t_1, ..., t_k \in T$, define $t \bowtie \{t_1, ..., t_k\}$ to mean that the transition t has the same connectivity as, collectively, the multiset of transitions, $\{t_1, ..., t_k\}$. Formally, $t \bowtie \{t_1, ..., t_k\}$ if and only if for all $n \in S \cup T$:

$$\begin{aligned} W(t, n) &= \sum_{i=1}^{k} W(t_i, n) \\ W(n, t) &= \sum_{i=1}^{k} W(n, t_i) \end{aligned}$$

For example, in Figure 2.8, $t_4 \bowtie \{t_1, t_6\}$, $t_5 \bowtie \{t_2, t_7\}$ and $t_4 \bowtie \{t_5\}$.

An equivalence relation, $\sim_{dpl}$, can be defined over the set of transitions, T, as follows:

$$\forall t_1, t_2 \in T : t_1 \sim_{dpl} t_2 \Leftrightarrow t_1 \bowtie \{t_2\}$$

The relation captures the notion of duplication of transitions, without any requirement that the transition labels are the same. Figure 2.8 shows the equivalence classes of transitions defined by $\sim_{dpl}$ for an implementation of the expression:

$$E = (((a \,\square\, \{\widehat{b}, c\}) \parallel (d \,\square\, d \,\square\, \{b, a\}))\ \mathbf{sy}\ b \,\square\, \widehat{a}); a$$

For a transition, $t \in T$, let $Dpl(t)$ be the equivalence class of $\sim_{dpl}$ to which t belongs:

$$Dpl(t) = \{t' \in T \mid t \sim_{dpl} t'\}$$

For example, in Figure 2.8, $Dpl(t_3) = \{t_1, t_2, t_3\}$ and $Dpl(t_8) = \{t_8\}$.

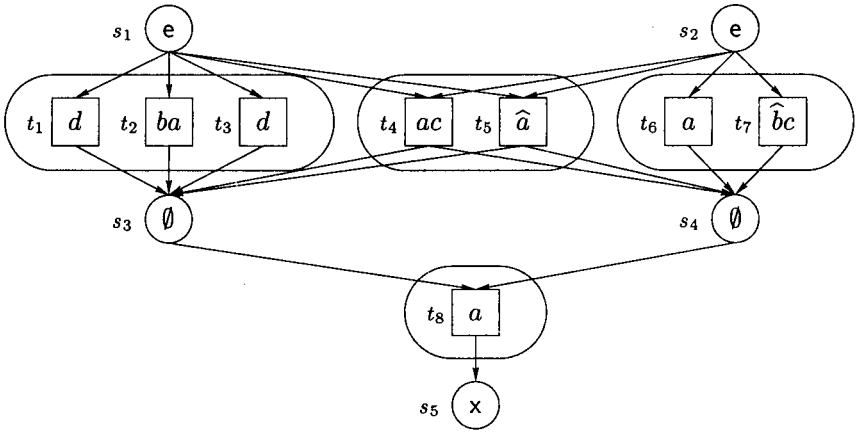


Figure 2.8: Duplication equivalence classes

By defining a total order over the transitions in the net, it is possible to define a unique or canonical representative for the equivalence class $Dpl(t)$, for each $t \in T$, given by $\min(Dpl(t))$. A total ordering over transitions, and the function, min, are defined in Section 2.5.6.

2.5.5 Synchronising transitions

In this section, a set of transitions is defined that is central to the synthesis algorithm for synchronisation, presented in Chapter 4. Although the definition is applicable to any net, it is only useful when the net has been derived from a box expression over a syntax that includes synchronisation, but not restriction, recursion or **stop** (see Table 4.1).

Let $\Sigma = (S, T, W, \lambda)$ be an implementation of a box expression from the syntax in Table 4.1. The set of transitions in Σ that have the same connectivity as a pair of transitions is given by:

$$T_{sc}(\Sigma) = \{t \in T \mid \exists t_1, t_2 \in T : t \bowtie \{t_1, t_2\}\}$$

For example, in Figure 2.8, $T_{sc} = \{t_4, t_5\}$, and in Figure 2.9, $T_{sc} = \{t_2, t_3\}$. The set of transitions, $T_{sc}(\Sigma)$ contains every transition in Σ that has arisen as a result of a synchronisation operation. An underlying net for Σ is obtained by removing the set of transitions $T_{sc}(\Sigma)$, giving the net $\Sigma \ominus T_{sc}(\Sigma)$. The remaining transitions, $T - T_{sc}(\Sigma)$ are known as the base transitions. Section 4.5 in Chapter 4 shows several useful properties associated with the definition of T_{sc}, including that $\Sigma \ominus T_{sc}(\Sigma)$ is the implementation of a box expression from the basic syntax and every transition in Σ arising from a synchronisation operation is a member of $T_{sc}(\Sigma)$.

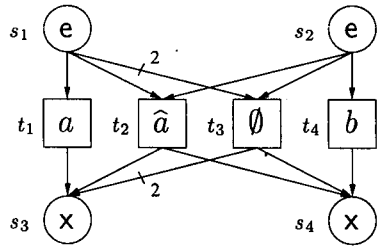


Figure 2.9: Synchronising transitions

Figure 2.10 shows implementations of the expressions:

$$\begin{aligned} E_1 &= a \parallel b \\ E_2 &= ((d \,\square\, d \,\square\, \{b, a\}) \parallel (a \,\square\, \{\widehat{b}, c\})); a \end{aligned}$$

Σ_1 and Σ_2 are isomorphic to the underlying nets of the nets in Figures 2.9, and 2.8, respectively.

For every transition, $t \in T$, a multiset of base transitions, $T_b(t)$, can be associated with t, such that $t \bowtie T_b(t)$. For all $t \in T$, $T_b(t)$ is defined as follows:

$$T_b(t) = \begin{cases} T_b(t_1) + T_b(t_2) & \text{if } t \in T_{sc}(\Sigma) \wedge t \bowtie \{t_1, t_2\} \\ \{\min(Dpl(t))\} & \text{otherwise} \end{cases}$$

For example, in Figure 2.9, $T_b(t_2) = \{t_1, t_4\}$, and $T_b(t_3) = \{t_1, t_1, t_4\}$.

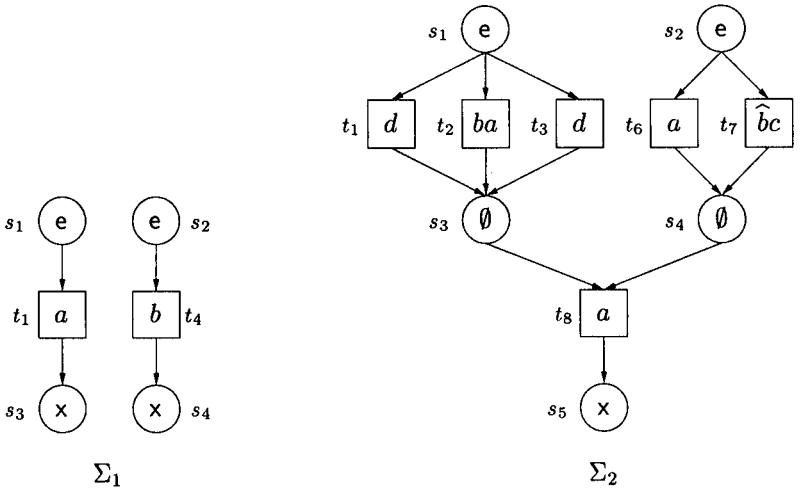


Figure 2.10: Base nets

2.5.6 Ordering of transitions

An ordering, $<_A$, over atomic actions can be defined. Let $<_b$ be any fixed ordering over the set of basic actions, $\mathcal{B}$. A unique word, $\mathcal{A}(\alpha) \in \mathcal{B}^*$ can be associated with each atomic action, α by writing the basic actions in α in order defined by $<_b$. For any atomic actions, α_1 and α_2:

$$\alpha_1 <_A \alpha_2 \Leftrightarrow \mathcal{A}(\alpha_1) <_{lex} \mathcal{A}(\alpha_2)$$

where $<_{lex}$ is a lexicographic ordering, using $<_b$.

For a finite net $\Sigma = (S, T, W, \lambda)$, let $<_t$ be an arbitrary fixed total order over the transitions in T. In general, where transition names are $t_1, t_2, ...$, it will be assumed that $t_i <_t t_j$ if and only if $i < j$. Hence a total order, $<_l$, of the set of transitions T, based on transition labels can be defined using $<_A$ and $<_t$:

$$\forall t_1, t_2 \in T : t_1 <_l t_2 \Leftrightarrow (\lambda(t_1) <_A \lambda(t_2)) \vee (\mathcal{A}(\lambda(t_1)) = \mathcal{A}(\lambda(t_2)) \wedge t_1 <_t t_2)$$

For a set of transitions, $T' \subseteq T$, define $\min(T')$ to be the smallest transition with respect to $<_l$:

$$\min(T') = t \in T' : \forall t' \in T' - \{t\}, t <_l t'$$

For example, for transitions $t_1 = \{d\}$, $t_2 = \{b, a\}$, $t_3 = \{d\}$, and $t_4 = \{a, c\}$, $t_2 <_l t_4 <_l t_1 <_l t_3$. Therefore, $\min(\{t_1, t_2, t_3, t_4\}) = t_2$, and $\min(\{t_1, t_3\}) = t_1$.

2.5.7 Actions and transitions

For any implementation of an expression from the basic syntax, shown in Table 2.3, there is a mapping from atomic actions in the expression to transitions in the implementation. The mapping is one-to-many – *i.e.* there may be several transitions associated with a single action. If the expression does not contain the iteration operator, then the mapping is one-to-one. In this section, the mapping between actions and transitions is formalised, and used to define an equivalence relation that relates transitions arising from the same atomic action.

$E ::=$	α	Atomic action
	$\mid E \parallel E$	Parallel composition
	$\mid E \,\square\, E$	Choice composition
	$\mid E; E$	Sequential composition
	$\mid [E * E * E]$	Iteration

Table 2.3: Basic box expression syntax

The representation of atomic actions is modified so as to consist of a set of atomic action names, together with a labelling function that associates a multiset of basic actions with each atomic action name. This representation is based on that used for transitions. For example, let $x_1, x_2, ..., x_5$ be a set of atomic action names, and μ a labelling function such that:

$$\mu(x_1) = \mu(x_3) = \mu(x_4) = a$$
$$\mu(x_2) = \mu(x_5) = \{a, b\}$$

An expression written using atomic action names, together with the labelling function, μ represents a unique expression from the standard notation. For

example, let:

$$E = (x_1 \parallel x_2) \,\square\, ((x_3; x_4) \parallel x_5) \tag{2.1}$$

then E, μ denotes the following standard notation expression:

$$E' = (a \parallel \{a, b\}) \,\square\, ((a; a) \parallel \{a, b\})$$

The purpose of such a representation for atomic actions is to distinguish actions which are the same multiset of basic actions. For example, x_2 and x_5 in E allow the two $\{a, b\}$ actions in E' to be distinguished.

Let E be an expression containing exactly the set of action names $X = \{x_1, ..., x_n\}$, μ a labelling function for X, and $\Sigma = \{S, T, W, \lambda\}$, an implementation of E, μ. Define $\phi : X \rightarrow 2^T$ to be a function such that for all $x \in X$, $\phi(x)$ is the set of transitions in Σ that have arisen from x. The function ϕ is defined inductively on the structure of E, and constructed for a particular implementation, Σ' of E, μ. The mapping can be extended from the particular implementation, Σ' to any implementation, Σ, by establishing an isomorphism between Σ' and Σ. Hence, the mapping is unique up to automorphism of Σ. Given E, μ, the function ϕ is obtained by constructing an implementation Σ' of E, μ, and recording the origin of each transition in Σ':

- $E = x$: The net $\Sigma' = (\{s_1, s_2\}, \{t\}, \{(s_1, t), (t, s_2)\}, \{(s_1, \mathsf{e}), (s_2, \mathsf{x}), (t, \mu(x))\})$ is an implementation of E. ϕ is defined by $\phi(x) = \{t\}$.

- $E = E_1 \parallel E_2$, $E = E_1 \,\square\, E_2$, $E = E_1; E_2$: Let Σ'_1 and Σ'_2 be disjoint implementations of E_1 and E_2, with mappings ϕ_1, and ϕ_2 respectively. Let Σ' be the implementation of E constructed from Σ'_1 and Σ'_2, and ϕ be the mapping between action names in E and sets of transitions in Σ'. ϕ is given by:

  $$\phi = \phi_1 \cup \phi_2$$

  The union operation for ϕ is defined below.

- $E = [E_1 * E_2 * E_3]$: Let Σ'_{ij} for $1 \leq i \leq 3$, $1 \leq j \leq 2$ be disjoint implementations of E_i, with mappings ϕ_{ij} respectively. Let Σ' be the

implementation of E constructed from the Σ'_{ij}, and ϕ be the mapping between action names in E and sets of transitions in Σ'. ϕ is defined by:

$$\phi = \bigcup_{\substack{1 \leq i \leq 3 \\ 1 \leq j \leq 2}} \phi_{ij}$$

For $1 \leq i \leq 3$, ϕ_{i1} and ϕ_{i2} have the same domain. Hence, the size of the set, $\phi(x)$ is doubled for each level of iteration that encloses x in the expression.

An auxiliary function, f is defined, which given an action name, $x \in X$, and a mapping ϕ returns $\emptyset$ if x is not in the domain of ϕ, and $\phi(x)$ otherwise:

$$f(x, \phi) = \begin{cases} \phi(x) & \text{if } x \in dom(\phi) \\ \emptyset & \text{otherwise} \end{cases}$$

The union of mapping functions, $\phi_1 \cup \phi_2$ is defined using the auxiliary function, f. For all $x \in X$:

$$(\phi_1 \cup \phi_2)(x) = f(x, \phi_1) \cup f(x, \phi_2)$$

An example construction of ϕ is given for the expression:

$$E = x_1 \,\square\, [x_2 * x_3 * x_4] \,\square\, x_5$$

where $\mu(x_1) = \mu(x_2) = \mu(x_5) = \{a\}$, $\mu(x_3) = \{b\}$, and $\mu(x_4) = \{\widehat{a}\}$. An implementation, Σ of E, μ is shown in Figure 2.11. The particular implementation, Σ' of E, μ, used in constructing ϕ, is chosen carefully so that there exists an isomorphism between Σ' and Σ that preserves transition names. By the inductive definition of ϕ:

$$\phi = \phi_1 \cup \phi_2 \cup \phi_3$$

where ϕ_i, for $1 \leq i \leq 3$ is a mapping from the subexpression E_i to the transitions in an implementation of E_i, with: $E_1 = x_1$, $E_2 = [x_2 * x_3 * x_4]$,

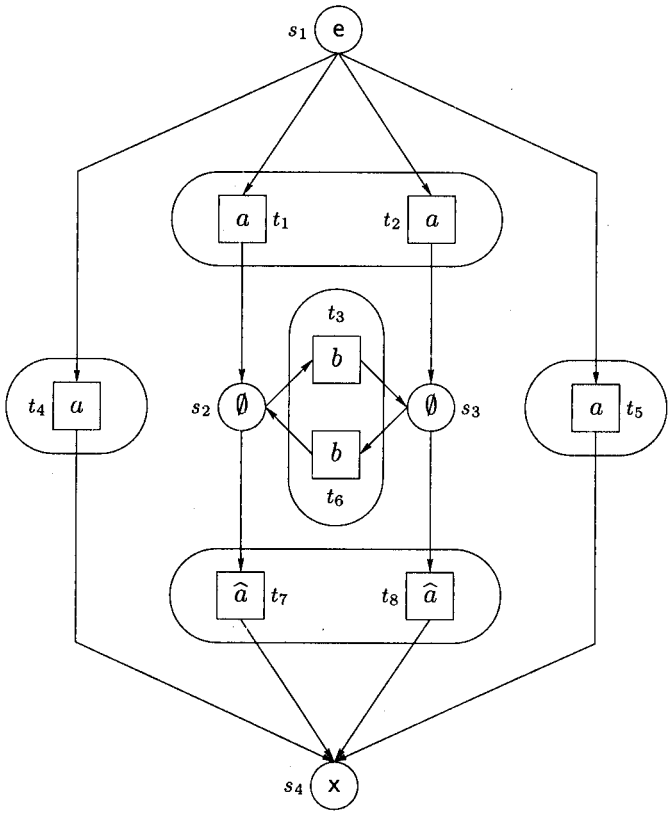


Figure 2.11: Equivalence classes of transitions arising from the same action

and $E_3 = x_5$. The implementations of E_1 and E_3 are chosen to contain the transitions t_4 and t_5 respectively. Hence ϕ_1 and ϕ_3 are defined by:

$$\begin{aligned} \phi_1(x_1) &= \{t_4\} \\ \phi_3(x_5) &= \{t_5\} \end{aligned}$$

ϕ_2 is defined inductively by $\phi_2 = \bigcup \phi_{ij}$, for $\leq i \leq 3$, $1 \leq j \leq 2$. Each ϕ_{ij} encodes the mapping between an atomic action and its implementation. The implementations are chosen so that the ϕ_{ij} are defined by:

$$\begin{aligned} \phi_{11}(x_2) &= \{t_1\} & \phi_{21}(x_3) &= \{t_3\} & \phi_{31}(x_4) &= \{t_7\} \\ \phi_{12}(x_2) &= \{t_2\} & \phi_{22}(x_2) &= \{t_6\} & \phi_{32}(x_4) &= \{t_8\} \end{aligned}$$

Hence, by the definition of the union of mapping functions, ϕ_2 is given by:

$$\phi_2(x_2) = \{t_1, t_2\} \qquad \phi_2(x_3) = \{t_3, t_6\} \qquad \phi_2(x_4) = \{t_7, t_8\}$$

Therefore, ϕ is defined by:

$$\begin{aligned} \phi(x_1) &= \{t_4\} & \phi(x_2) &= \{t_1, t_2\} & \phi(x_3) &= \{t_3, t_6\} \\ \phi(x_4) &= \{t_7, t_8\} & \phi(x_5) &= \{t_5\} \end{aligned}$$

Given a mapping, ϕ, between the set of action names, X, in an expression, and the set of transitions, T, in an implementation of the expression, an equivalence relation, $\sim_\phi$, can be defined by:

$$\forall t_1, t_2 \in T : t_1 \sim_\phi t_2 \Leftrightarrow \exists x \in X \text{ such that } \{t_1, t_2\} \subseteq \phi(x)$$

Define $\phi(t)$ to be the equivalence class of $\sim_\phi$ to which the transition t belongs:

$$\forall t \in T : \phi(t) = \{t' \in T \mid t \sim_\phi t'\}$$

Figure 2.11 indicates the equivalence classes of $\sim_\phi$ for the mapping ϕ constructed above. In Chapter 4, Section 4.3 describes an algorithm for constructing the set of equivalence classes of $\sim_\phi$ for any implementation of a box expression from the basic syntax.

2.5.8 The $\odot$ operator

The $\odot$ operator is used to relate the synchronisation of actions in expressions with the sets of transitions that are created in the corresponding net. This is achieved using the equivalence classes of $\sim_\phi$. Define an auxiliary function, θ, which allows each parameter of $\odot$ to be either a multiset, or a set of multisets:

$$\theta(t) = \begin{cases} \{t\} & \text{if } t \in T \\ t & \text{otherwise} \end{cases}$$

Then, $\odot$ is defined by:

$$T_1 \odot T_2 = \{\theta(t_1) + \theta(t_2) \mid t_1 \in T_1, t_2 \in T_2\}$$

The $\odot$ operator produces a set of multisets, rather than a multiset of multisets. Hence, there is only one copy of, for example, $\{t_4, t_5, t_7\}$ in:

$$\begin{aligned} (\{t_4, t_5\} \odot \{t_6, t_7\}) \odot \{t_4, t_5\} &= \{\{t_4, t_6\}, \{t_4, t_7\}, \{t_5, t_6\}, \{t_5, t_7\}\} \odot \{t_4, t_5\} \\ &= \{\{t_4, t_4, t_6\}, \{t_4, t_5, t_6\}, \{t_4, t_4, t_7\}, \\ &\quad \{t_4, t_5, t_7\}, \{t_5, t_5, t_6\}, \{t_5, t_5, t_7\}\} \end{aligned}$$

Using the example construction for ϕ for the expression, E, (2.1) and the net in Figure 2.11, the sets of transitions that would be produced by the synchronisation operation in E **sy** a can be found using the $\odot$ operator. The pairs of actions in E which synchronise are (x_1, x_4),(x_2, x_4) and (x_4, x_5). The corresponding sets of transitions generated by the three synchronisations are:

$$\begin{aligned} \phi(x_1) \odot \phi(x_4) &= \{t_4\} \odot \{t_7, t_8\} \\ &= \{\{t_4, t_7\}, \{t_4, t_8\}\} \\ \phi(x_2) \odot \phi(x_4) &= \{t_1, t_2\} \odot \{t_7, t_8\} \\ &= \{\{t_1, t_7\}, \{t_1, t_8\}, \{t_2, t_7\}, \{t_2, t_8\}\} \\ \phi(x_4) \odot \phi(x_5) &= \{t_7, t_8\} \odot \{t_5\} \\ &= \{\{t_5, t_7\}, \{t_5, t_8\}\} \end{aligned}$$

Hence, for example, the synchronisation of the actions x_1 and x_4 generates two transitions t'_1 and t'_2, such that $t'_1 \bowtie \{t_4, t_7\}$, $t'_2 \bowtie \{t_4, t_8\}$.

Chapter 3

Basic synthesis

3.1 Introduction

$E ::=$	α	Atomic action
	$\mid E \parallel E$	Parallel composition
	$\mid E \,\square\, E$	Choice composition
	$\mid E; E$	Sequential composition
	$\mid [E * E * E]$	Iteration

Table 3.1: Basic box expression syntax

Table 3.1 contains a subset of the full box expression syntax given in Table 1.1 and [6]. The syntax in Table 3.1, christened the basic box expression syntax, was chosen for an initial investigation into the synthesis and axiomatisation problems because the operators in Table 3.1 preserve a strong correspondence between the structure of a net, and the structure of the expression from which that net is derived. This correspondence would not be so strong if operators such as synchronisation and restriction were included. For the remainder of this chapter, every box expression should be assumed to be a member of the language generated by the syntax in Table 3.1, unless otherwise stated.

Sections 3.2 and 3.3 present a solution to the synthesis problem for the

class of input nets that can be obtained from an expression over the syntax in Table 3.1. In Section 3.4, the correctness of the synthesis algorithm is shown. The detailed analysis carried out in Section 3.4 forms the basis for the discussion, and solutions to related problems presented in Section 3.5. A canonical form for box expressions is defined in Section 3.5. This allows the synthesis algorithm, described in Sections 3.2 and 3.3 to be modified to synthesise canonical form expressions, and provides a basis for the derivation of a complete axiom system for the fragment of the Petri Box Calculus given in Table 3.1. Section 3.5 concludes with an investigation into the possibility of the automatic generation of proofs of equivalence for expressions from the syntax in Table 3.1.

3.2 The synthesis algorithm

The synthesis algorithm takes as input a net, Σ, which is an implementation of some unknown box expression. The output is a box expression, E, such that Σ is an implementation of E. The algorithm is based on a set of synthesis rules, with one rule for each operator in the box expression syntax of Table 3.1. Each synthesis rule has a set of preconditions which must hold for the rule to be applied. These conditions are based on the structural properties of nets described in Section 2.5 in Chapter 2. When a synthesis rule is applied, the input net is decomposed into a collection of subnets, and at the same time, the expression corresponding to the input net is refined. The rules are applied recursively to each subnet obtained by net decomposition, until the expression is fully refined.

A tree data structure is used by the synthesis algorithm. A node of the tree has the form shown in Figure 3.1. Initially, only the net field of the node contains any data. Once the preconditions of the synthesis rules have been checked, and the rule to apply has been identified, the type field of the node is set. The type field is used to indicate which synthesis rule is to be applied.

If the rule to be applied is the atomic action rule, then the node is a leaf node of the tree, and the action field is set to be the atomic action synthesised from the input net. Otherwise, the node is internal, and the list field is used to point to a collection of children, with each child node containing a subnet obtained by the net decomposition of the synthesis rule. Once the synthesis process has completed, the tree structure can be interpreted to obtain the synthesised expression. For this interpretation, only the type and action/list fields of the nodes are required - the net field can be ignored.

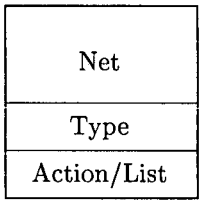


Figure 3.1: Data structure of a node

The pseudo-code for the synthesis algorithm is given below. BOX EXPRESSION SYNTHESIS creates a root node, and initialises the net field with the input net. SYNTHESISE is a recursive procedure which takes as input the root node of a (sub)tree, and expands it into a tree structure corresponding to an expression for the net at the root node. EXPRESSION performs a depth first traversal of the fully expanded tree, and uses the type and action fields of the nodes to find the synthesised expression.

```
BOX EXPRESSION SYNTHESIS(Σ)
1   N=new node
2   N.net=Σ
3   SYNTHESISE(N)
4   return EXPRESSION(N)

SYNTHESISE(N)
```

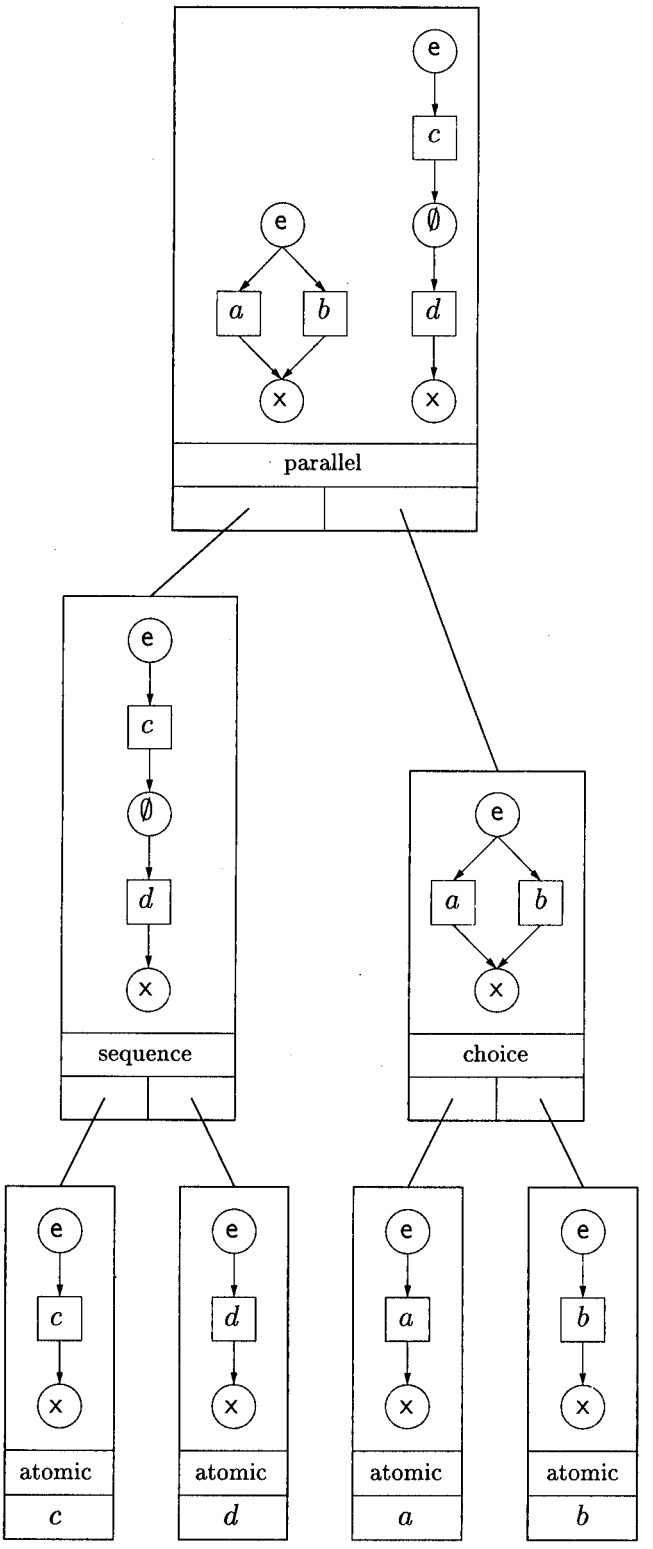


Figure 3.2: Tree produced by the synthesis algorithm

```
1   N.type=ANALYSE(N.net)
2   case N.type
3         atomic: ATOMIC(N)
4         parallel: PARALLEL(N)
5         choice: CHOICE(N)
6         iteration: ITERATION(N)
7         sequence: SEQUENCE(N)
8   for each node N' in N.list
9       do SYNTHESISE(N')
```

For example, given the net, Σ, in Figure 2.6, the call to BOX EXPRESSION SYNTHESIS(Σ) will construct the tree shown in Figure 3.2. Performing a depth first traversal of this tree obtains the expression $(c; d) \parallel (a \,\square\, b)$. Note that in general, the tree will not be binary.

3.2.1 Preconditions

The preconditions of the synthesis rules are based on four structural properties of nets:

1. **Number of transitions:** This property is true if there is more than one transition in the net, and false if there are zero or one transitions.

$$Pr_1 = |T| > 1$$

2. **Connectedness:** This property is true if there is an undirected path between every pair of nodes in the net, and false if the net consists of at least two disjoint components.

$$Pr_2 = \forall n_1, n_2 \in N_{\mathsf{a}} : n_1 \overset{\leftrightarrow}{\sim}_{N_{\mathsf{a}}} n_2$$

3. **Internal connectedness:** This property considers the connectedness of the net when all entry and exit places are removed. The property is

true if the net is connected after deleting the entry and exit places, and false if there are at least two disjoint internal components.

$$Pr_3 = \forall n_1, n_2 \in N_{\mathsf{i}} : n_1 \overleftrightarrow{\sim}_{N_{\mathsf{i}}} n_2$$

4. **Internal Interface:** This property is true if there is no undirected path from an entry place to an exit place, when some cluster of internal places is removed.

$$Pr_4 = \exists s \in S_{\mathsf{i}} : \forall s_1 \in S_{\mathsf{e}}, s_2 \in S_{\mathsf{x}} : s_1 \overleftrightarrow{\not\sim}_{(N_{\mathsf{a}} - \mathcal{C}(s))} s_2$$

Type	Property 1	Property 2	Property 3	Property 4
Atomic action	false	true	true	false
Parallel	true	false	false	false
Choice	true	true	false	false
Iteration	true	true	true	false
Sequence	true	true	true	true

Table 3.2: Preconditions for the synthesis rules

Table 3.2 shows which properties hold for each type of expression. Figure 3.3 illustrates how the four properties provide a simple decision procedure, to identify the synthesis rule to apply. This procedure is implemented by the ANALYSE function, given below. Using this approach, all four properties of the net are only tested in the worst case, when the sequence or iteration rule is to be applied.

```
ANALYSE(Σ)
1   if Pr1 holds for Σ
2     then if Pr2 holds for Σ
3               then if Pr3 holds for Σ
```

4 **then if** Pr_4 holds for Σ

5 **then return** sequence

6 **else return** iteration

7 **else return** choice

8 **else return** parallel

9 **else return** atomic

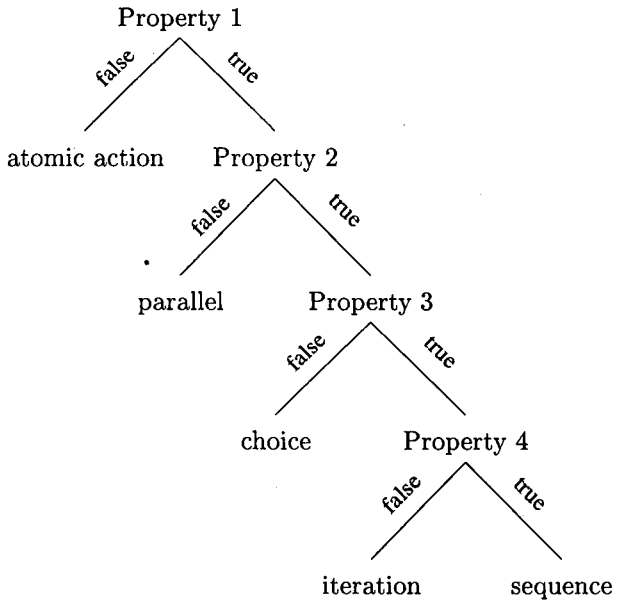


Figure 3.3: Decision procedure for identifying synthesis rule to apply

3.3 Synthesis rules

This section describes the five synthesis rules that are used by the algorithm. These correspond to the procedures, ATOMIC, PARALLEL, CHOICE, ITERATION and SEQUENCE, called by the SYNTHESISE procedure. For each rule, the method for expression refinement and net decomposition is given, followed by optional checks that can be carried out to ensure that the input net is the implementation of a box expression. These checks allow the synthesis algorithm to be used to recognise the class of nets that can be derived from

expressions over the syntax in Table 3.1. Finally, an example illustrating the use of the rule is given. The expression refinement involves determining the number of components that the input net should be decomposed into. The examples show how a node of the tree data structure, described in Section 3.2, is expanded by a rule application. Each example has been chosen so that the preconditions of the particular rule are satisfied.

The tree structure that is constructed by the synthesis algorithm does not correspond directly to a parse tree of the synthesised expression. For example, an implementation of $E = (a \parallel b) \parallel (c \parallel d)$ will be synthesised to a tree containing a root node with four child leaf nodes. Advantage is taken of the associativity of the parallel, choice and sequence operators, which means that E can be written unambiguously as $a \parallel b \parallel c \parallel d$. The EXPRESSION function produces a properly bracketed expression from the tree by imposing a right-associative bracketing order, giving the expression $a \parallel (b \parallel (c \parallel d))$, for E.

A further operator on labelled nets, $\uplus$, is defined. This operator is used in the decomposition of the input net to the synthesis algorithm by some of the synthesis rules in this section. Let $\Sigma = (S, T, W, \lambda)$ be a labelled net, P be a set of new places, and $l \in \{\mathsf{e}, \emptyset, \mathsf{x}\}$ be the label which is to be assigned to the places in P. Each place, $p \in P$ has the form (T_1, T_2), where $T_1 \subseteq T$ is the set of transitions which have an arc to p, and $T_2 \subseteq T$ is the set of transitions which have an arc from p. The net, $\Sigma \uplus (P, l)$, obtained by adding the set of new places to Σ is defined by:

$$\Sigma \uplus (P, l) = (S \cup P, T, W', \lambda')$$

where $W' : (S \cup P \cup T) \times (S \cup P \cup T) \rightarrow \{0, 1\}$, and λ' are defined as follows:

$$W'(n_1, n_2) = \begin{cases} W(n_1, n_2) & \text{if } n_1, n_2 \in S \cup T \\ 1 & \text{if } n_1 = (T_1, T_2) \in P, n_2 \in T_2 \\ 1 & \text{if } n_1 \in T_1, n_2 = (T_1, T_2) \in P \\ 0 & \text{otherwise} \end{cases}$$

$$\lambda'(n) = \begin{cases} \lambda(n) & \text{if } n \in S \cup T \\ l & \text{if } n \in P \end{cases}$$

In Section 1.3.5, a general form for the semantics of the parallel, choice, sequence and iteration operators was described. There is a corresponding general approach that can be used for the net decomposition performed by the synthesis rules for these operators. Let Σ be a net which is known to have arisen from a particular semantic rule. The $\uplus$ operator can be used in a general technique for decomposing Σ into its subnets, Σ_i for $1 \leq i \leq k$ for some k:

1. The sets of transitions belonging to each subnet, Σ_i are identified in Σ.

2. The clusters of places, corresponding to the interfaces between the Σ_i components in Σ are identified. Each cluster is decomposed into sets of new places corresponding to the original entry and exit interfaces of the subnets. These sets of places have the form used by the $\uplus$ operator.

3. The clusters of places identified in 2 are removed, using the $\ominus$ operator, and the sets of new places are added to the resulting net using the $\uplus$ operator.

4. The net, Σ', obtained in 3 corresponds to the disjoint union of the subnets Σ_i. For $1 \leq i \leq k$, the subnet Σ_i can be obtained from Σ', using $\Sigma' \mid_{\mathcal{G}(T_i)}$, where T_i is the set of transitions belonging to Σ_i, identified in step 1.

In the descriptions of the synthesis rules, the following conventions should be assumed: Σ is the input net to be decomposed, and E is the unrefined expression corresponding to Σ. The subnets obtained by decomposing Σ are Σ_i, with corresponding unrefined expressions E_i, for $1 \leq i \leq k$ (for some $k > 1$).

3.3.1 Atomic action

The synthesis rule for atomic actions is the only base case rule – *i.e.* the recursion of the SYNTHESISE procedure ends with an application of the atomic action rule. No net decomposition is performed, and the refined expression contains no free variables.

Expression refinement

The expression is refined to be the label of the single transition in the net:

$$E = \lambda(t) \text{ where } T = \{t\} \tag{3.1}$$

Optional checks

There should be exactly two places in Σ, one an entry place, with an arc to t, and the other an exit place with an arc from t. These additional checks ensure that the input net is an implementation of an atomic action.

Example

Figure 3.4 shows an implementation of the expression $E = \{a\}$. Applying the atomic action synthesis rule, (3.1), to this net gives the fully refined expression $E = \{a\}$.

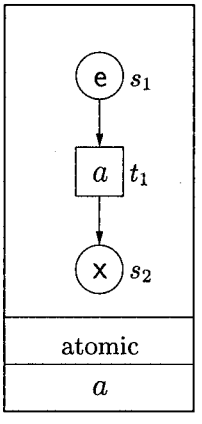


Figure 3.4: Atomic action

3.3.2 Parallel composition

Expression refinement

The equivalence classes of the relation $\overset{\leftrightarrow}{\sim}_{N_\mathsf{a}}$ partition the net Σ into its disjoint components. Let C be the set of equivalence classes and n be the cardinality of C. Hence n is the number of disjoint components in Σ. The expression is refined to:

$$E = E_1 \parallel E_2 \parallel ... \parallel E_n \tag{3.2}$$

Net decomposition

Let $C_1, ..., C_n$ be the equivalence classes in C – *i.e.* $C = \{C_1, C_2, ..., C_n\}$. For $1 \leq i \leq n$:

$$\Sigma_i = \Sigma \mid_{C_i} \tag{3.3}$$

Optional checks

No additional checks are required for this synthesis rule.

Example

Figure 3.5 shows an implementation of the expression $a \parallel ((b \,\square\, c) \parallel (d; e))$. This net is disjoint, therefore, by Table 3.2, the parallel composition synthesis rule is applicable. The set of equivalence classes, C, given by the relation $\overset{\leftrightarrow}{\sim}_{N_\mathsf{a}}$ is:

$$C = \{\{s_3, t_2, t_3, s_4\}, \{s_5, t_4, s_6, t_5, s_7\}, \{s_1, t_1, s_2\}\}$$

Therefore, $n = 3$, and by (3.2), E is refined to:

$$E = E_1 \parallel E_2 \parallel E_3$$

The decomposed nets, Σ_1, Σ_2, and Σ_3 are shown in Figure 3.5. Using (3.3) with, for example, $C_3 = \{s_1, t_1, s_2\}$ gives, by (3.3):

$$\Sigma_3 = (\{s_1, s_2\}, \{t_1\}, \{(s_1, t_1), (t_1, s_2)\}, \{(s_1, \mathsf{e}), (s_2, \mathsf{x}), (t_1, \{a\})\})$$

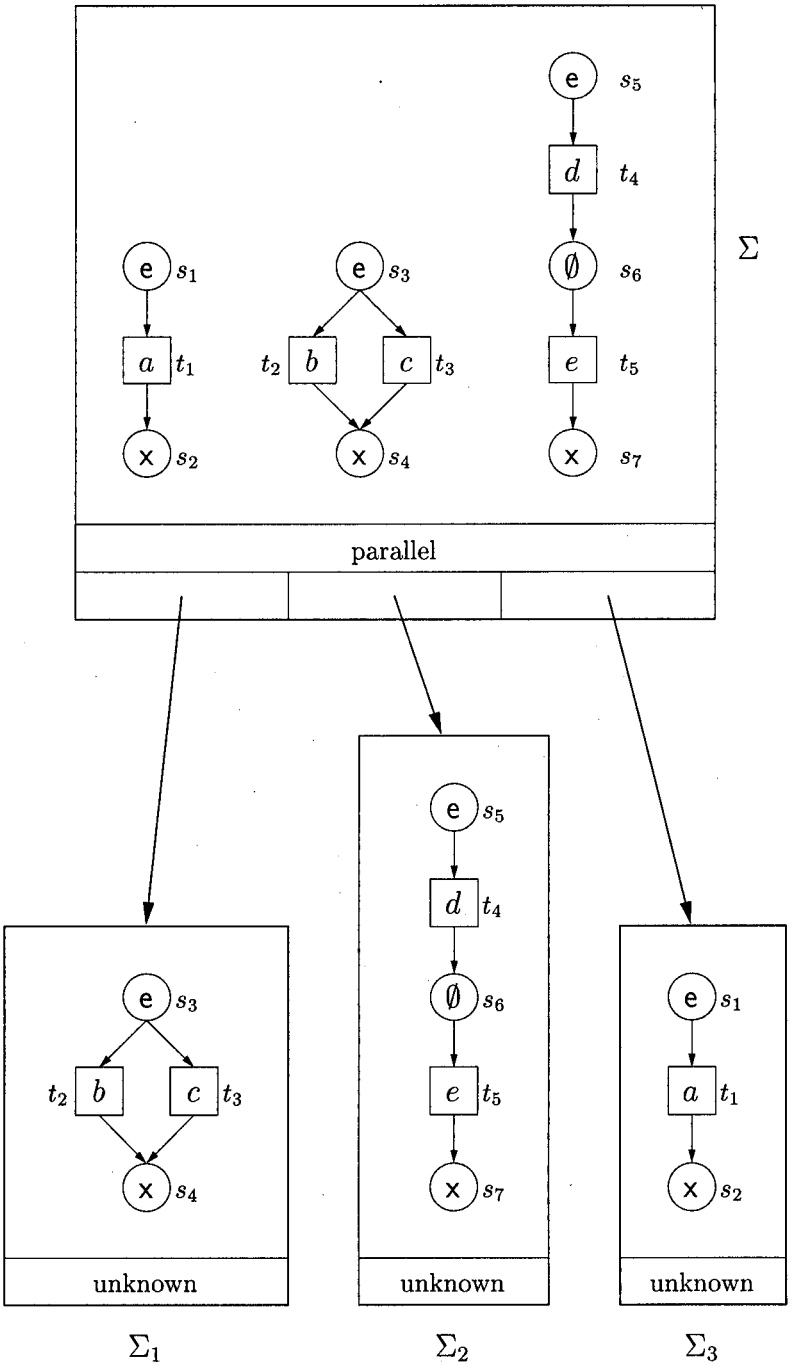


Figure 3.5: Parallel composition

3.3.3 Choice composition

Expression refinement

Define $\sim_{||}$, a relation over the set of entry transitions, T_{e}, of Σ, such that two transitions are related if there is no entry place with an arc to both of them.

$$\forall t_1, t_2 \in T_{\mathsf{e}} : t_1 \sim_{||} t_2 \Leftrightarrow \forall s \in S_{\mathsf{e}} : W(s, t_1) = 0 \vee W(s, t_2) = 0$$

Let $\sim_{||}^*$ be the transitive closure of $\sim_{||}$. A relation, $\sim_{\mathsf{e}}$ is defined over the set of entry transitions:

$$\forall t_1, t_2 \in T_{\mathsf{e}} : t_1 \sim_{\mathsf{e}} t_2 \Leftrightarrow t_1 \sim_{||}^* t_2 \vee t_1 \overset{\leftrightarrow}{\sim}_{N_{\mathsf{I}}} t_2$$

Proposition 11, proved in Section 3.4, shows that $\sim_{\mathsf{e}}$ is an equivalence relation whenever Σ is an implementation of a choice expression. Let $P_{T_{\mathsf{e}}}$ be the set of equivalence classes of T_{e}, formed by the relation $\sim_{\mathsf{e}}$. Corollary 2, in Section 3.4, shows that a corresponding set of equivalence classes of the exit transitions, T_{x} can be given by:

$$P_{T_{\mathsf{x}}} = \{\{t \in T_{\mathsf{x}} \mid \exists t' \in P : t \overset{\leftrightarrow}{\sim}_{N_{\mathsf{I}}} t'\} \mid P \in P_{T_{\mathsf{e}}}\} \tag{3.4}$$

Let n be the number of equivalence classes of $P_{T_{\mathsf{e}}}$ – *i.e.* $n = |P_{T_{\mathsf{e}}}|$. The expression, E is refined to:

$$E = E_1 \,\square\, E_2 \,\square\, \ldots \,\square\, E_n \tag{3.5}$$

Net decomposition

The equivalence classes $P_{T_{\mathsf{e}}}$, and $P_{T_{\mathsf{x}}}$ are used to decompose the entry and exit interfaces of Σ as follows:

$$\begin{aligned} X_{\mathsf{e}} &= \{(\emptyset, s^\bullet \cap P) \mid P \in P_{T_{\mathsf{e}}}, s \in S_{\mathsf{e}}\} \\ X_{\mathsf{x}} &= \{({}^\bullet s \cap P, \emptyset) \mid P \in P_{T_{\mathsf{x}}}, s \in S_{\mathsf{x}}\} \end{aligned} \tag{3.6}$$

An auxiliary net, Σ_a, is constructed by removing the entry and exit interfaces of Σ, and adding the interfaces defined by (3.6):

$$\Sigma_a = \Sigma \uplus (X_{\mathsf{e}}, e) \uplus (X_{\mathsf{x}}, x) \ominus (S_{\mathsf{e}} \cup S_{\mathsf{x}}) \tag{3.7}$$

Σ_a is an implementation of the parallel composition of the decomposed nets $\Sigma_1, ..., \Sigma_n$. It is not necessarily true that there will be n disjoint components – there will be more if any Σ_i is the implementation of a parallel composition expression. The set of equivalence classes $P_{T_{\mathsf{e}}} = \{P_1, P_2, ..., P_n\}$ (or, equally $P_{T_{\mathsf{x}}}$), can be used to decompose Σ_a into $\Sigma_1, ..., \Sigma_n$.

$$\Sigma_i = \Sigma_a \mid_{\mathcal{G}(P_i)} \text{ for } 1 \leq i \leq n \tag{3.8}$$

Optional checks

If the input net is the implementation of a box expression, then the sets of equivalence classes, $P_{T_{\mathsf{e}}}$ and $P_{T_{\mathsf{x}}}$ will be such that $|P_{T_{\mathsf{e}}}| = |P_{T_{\mathsf{x}}}|$ and $|P_{T_{\mathsf{e}}}| > 1$. If the sets of equivalence classes do not satisfy these properties, then the synthesis algorithm has detected that the input net is not an implementation of a box expression.

It remains to check that the decomposition of the entry and exit interfaces of Σ is valid. This is achieved by recombining the decomposed interfaces and checking that they match the original interfaces in Σ. The set of new places, X_{e} can be partitioned into $X_1, ..., X_n$, where $n = |P_{T_{\mathsf{e}}}|$ according to the equivalence class of $P_{T_{\mathsf{e}}}$ that each place in X_{e} arises from. The places in X_{e} have the form $(\emptyset, T')$, where T' is a set of transitions belonging to Σ. In order to check that the decomposition is valid, it is necessary to combine the sets of places $X_1, ..., X_n$, and match the result against $S_{\mathsf{e}}(\Sigma)$. Since X_{e} is a set of places, rather than a multiset, it is sufficient to check that $|X_1| \cdot |X_2| \cdot ... \cdot |X_n| = |S_{\mathsf{e}}(\Sigma)|$, and for any set of places $(\emptyset, T_i) \in X_i$ for $1 \leq i \leq n$, there exists a place $s \in S_{\mathsf{e}}(\Sigma)$ such that $s^\bullet = T_1 \cup ... \cup T_n$. A similar procedure is used to check that the set of new places, X_{x} is a valid decomposition of the exit interface of Σ.

Example

Figure 3.6 shows an implementation of the expression

$$(a \parallel (b \,\square\, c)) \,\square\, (d; (e \,\square\, f)) \,\square\, (g \parallel h)$$

This net is connected, but not internally connected – for example $t_1 \overleftrightarrow{\gamma}_{N_1} t_2$. Therefore, the choice composition synthesis rule is applicable.

The sets of entry and exit transitions are $T_{\mathsf{e}} = \{t_1, ..., t_6\}$, and $T_{\mathsf{x}} = \{t_1, ..., t_5, t_7, t_8\}$ respectively. The set of equivalence classes of T_{e}, given by the relation $\sim_{\parallel}^{*}$ is $\{\{t_1, t_4, t_5\}, \{t_6\}, \{t_2, t_3\}\}$. There is no internal path between any pair of transitions in T_{e}. Therefore, the equivalence classes of the entry and exit transitions are:

$$P_{T_{\mathsf{e}}} = \{\{t_1, t_4, t_5\}, \{t_2, t_3\}, \{t_6\}\}$$
$$P_{T_{\mathsf{x}}} = \{\{t_1, t_4, t_5\}, \{t_2, t_3\}, \{t_7, t_8\}\}$$

Hence, by (3.5), the expression E is refined to:

$$E = E_1 \,\square\, E_2 \,\square\, E_3$$

The new entry and exit interfaces, defined by (3.6) are:

$$\begin{aligned} X_{\mathsf{e}} &= \{s_{10} = (\emptyset, \{t_1\}), s_{11} = (\emptyset, \{t_4, t_5\}), s_{12} = (\emptyset, \{t_2\}), s_{13} = (\emptyset, \{t_3\}), \\ &\quad s_{14} = (\emptyset, \{t_6\})\} \\ X_{\mathsf{x}} &= \{s_{15} = (\{t_1\}, \emptyset), s_{16} = (\{t_4, t_5\}, \emptyset), s_{17} = (\{t_2\}, \emptyset), s_{18} = (\{t_3\}, \emptyset), \\ &\quad s_{19} = (\{t_7, t_8\}, \emptyset)\} \end{aligned}$$

Where $s_{10}, ..., s_{19}$ are introduced as shorthand for the identifiers of the new places – *i.e.* the real place identifier for the place labelled s_{10} in Figure 3.6 is $(\emptyset, \{t_1\})$.

When the entry and exit interfaces of Σ are removed, and the entry and exit interfaces given by X_{e} and X_{x} are added, the net $\Sigma_a = \Sigma_1 \sqcup \Sigma_2 \sqcup \Sigma_3$ is obtained. The disjoint components of Σ_a are related according to the three equivalence classes of $P_{T_{\mathsf{x}}}$, decomposing Σ_a into Σ_1, Σ_2 and Σ_3. For example,

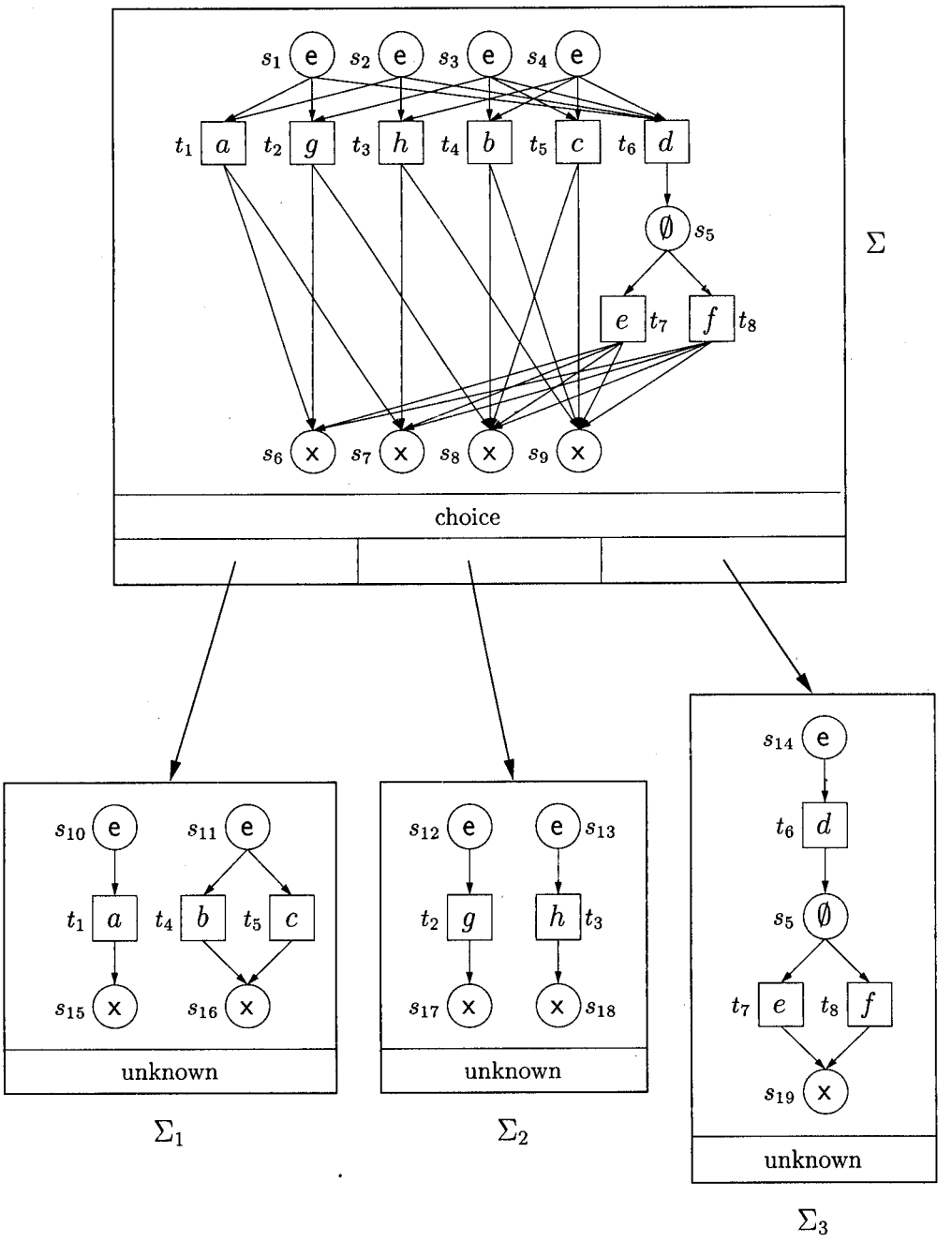


Figure 3.6: Choice composition

by (3.8), using the equivalence class $\{t_2, t_3\}$:

$$\begin{aligned}\Sigma_2 &= \Sigma_a \mid_{\mathcal{G}(\{t_2,t_3\})} \\ &= (\{s_{12}, s_{13}, s_{17}, s_{18}\}, \{t_2, t_3\}, \{(s_{12}, t_2), (s_{13}, t_3), (t_2, s_{17}), (t_3, s_{18})\}, \\ &\quad \{(s_{12}, \mathsf{e}), (s_{13}, \mathsf{e}), (s_{17}, \mathsf{x}), (s_{18}, \mathsf{x}), (t_2, \{g\}), (t_3, \{h\})\})\end{aligned}$$

since $\mathcal{G}(\{t_2, t_3\}) = \{s_{12}, s_{13}, s_{17}, s_{18}, t_2, t_3\}$.

3.3.4 Sequence

Expression refinement

Define $S_;$ to be the set of clusters of internal places such that for each cluster, $c \in S_;$, there is no undirected path between an entry and exit place in the net $\Sigma \ominus c$:

$$S_; = \{c \in \mathcal{C}_{\mathsf{i}}(\Sigma) \mid \forall s_1 \in S_{\mathsf{e}}, s_2 \in S_{\mathsf{x}} : s_1 \overleftrightarrow{\not\sim}_{(N_{\mathsf{a}}-c)} s_2\} \tag{3.9}$$

$n = |S_;| + 1$ is the number of subnets that will be formed when the net is decomposed. Therefore, the expression, E, is refined to:

$$E = E_1; E_2; ...; E_n \tag{3.10}$$

Net decomposition

Removing any cluster of places $c \in S_;$ partitions Σ into two components – one component consists of the collection of nodes connected to some entry place, and the other contains the nodes connected to some exit place. The function $C_{\mathsf{e}}(c)$ is defined to give the set of nodes in the component containing the entry places:

$$C_{\mathsf{e}}(c) = \{n' \in S \cup T \mid \exists s \in S_{\mathsf{e}} : s \overleftrightarrow{\sim}_{(N_{\mathsf{a}}-c)} n'\} \tag{3.11}$$

A total order, $<_s$, over the clusters of places in $S_;$ can be defined. For $c_1, c_2 \in S_;$:

$$c_1 <_s c_2 \Leftrightarrow |C_{\mathsf{e}}(c_1)| < |C_{\mathsf{e}}(c_2)|$$

$<_s$ is used to obtain an order $c_1, c_2, ..., c_{(n-1)}$ of the clusters of places in $S_;$. The auxiliary nets Σ'_i, for $1 \leq i \leq n$, are obtained by removing the clusters of places in $S_;$ from Σ:

$$\Sigma'_i = \begin{cases} \Sigma \mid_{C_{\mathsf{e}}(c_1)} & \text{if } i = 1 \\ \Sigma \mid_{(C_{\mathsf{e}}(c_i) - (C_{\mathsf{e}}(c_{i-1}) \cup c_{i-1}))} & \text{if } 1 < i < n \\ \Sigma \mid_{(N_{\mathsf{a}} - (C_{\mathsf{e}}(c_{(n-1)}) \cup c_{n-1}))} & \text{if } i = n \end{cases} \tag{3.12}$$

Two functions, one for the entry interface (I_{e}), and one for the exit interface (I_{x}) are defined. These functions decompose a cluster of places $c' \in S_;$, into the corresponding entry or exit interface:

$$\begin{aligned} I_{\mathsf{e}}(c') &= \{(\emptyset, s^\bullet) \mid s \in c'\} \\ I_{\mathsf{x}}(c') &= \{({}^\bullet s, \emptyset) \mid s \in c'\} \end{aligned} \tag{3.13}$$

The decomposed subnets $\Sigma_1, .., \Sigma_n$ are obtained by adding new interfaces to the nets $\Sigma'_1, .., \Sigma'_n$, defined by (3.12). The new interfaces are generated by applying the interface functions I_{e} and I_{x} to the clusters $c_1, c_2, ..., c_{n-1} \in S_;$:

$$\Sigma_i = \begin{cases} \Sigma'_1 \uplus (I_{\mathsf{x}}(c_1), x) & \text{if } i = 1 \\ \Sigma'_i \uplus (I_{\mathsf{e}}(c_{i-1}), e) \uplus (I_{\mathsf{x}}(c_i), x) & \text{if } 1 < i < n \\ \Sigma'_n \uplus (I_{\mathsf{e}}(c_{n-1}), e) & \text{if } i = n \end{cases} \tag{3.14}$$

Optional checks

If the input net is the implementation of a box expression, then $S_;$ is guaranteed to contain at least one cluster of places. If $S_; = \emptyset$, then the synthesis algorithm has detected that the input net is not an implementation of a box expression.

It remains to check that the decomposition of each cluster in $S_;$ is valid. This is achieved by recombining the decomposed interfaces and checking that they match the original clusters in Σ. For each cluster, $c \in S_;$, $I_{\mathsf{e}}(c)$ ($I_{\mathsf{x}}(c)$) are sets of pairs of sets of transitions, such that the first (second) set of transitions

in the pair is $\emptyset$. Since $I_{\mathsf{e}}(c)$ and $I_{\mathsf{x}}(c)$ are sets, rather than multisets, it is sufficient to check that for each cluster $c \in S_;$, $|I_{\mathsf{x}}(c)| \cdot |I_{\mathsf{e}}(c)| = |c|$ and:

$$\forall (T_1, \emptyset) \in I_{\mathsf{x}}(c), (\emptyset, T_2) \in I_{\mathsf{e}}(c) : \exists s \in c \text{ such that } {}^\bullet s = T_1 \wedge s^\bullet = T_2$$

If a cluster does not match the recomposition of the decomposed interfaces, then the net decomposition was not valid. This implies that the input net, Σ, is not the implementation of a box expression.

Example

Figure 3.7 shows the net, Σ which is an implementation of the expression:

$$a; ((b; c) \,\square\, d); (e \parallel (f; g))$$

Σ is connected, and also internally connected. Removing the place s_2 leaves no path between an entry and exit place. Therefore, the sequential composition synthesis rule is applicable. The set of internal clusters is given by $S' = \{\{s_2\}, \{s_3\}, \{s_4, s_5\}, \{s_7\}\}$. Therefore, by (3.9):

$$S_; = \{\{s_2\}, \{s_4, s_5\}\}$$

The net, Σ will be decomposed into three subnets, because there are two interface clusters in $S_;$. Therefore, by (3.10), the expression E is refined to:

$$E = E_1; E_2; E_3$$

The two clusters of places in $S_;$ are ordered $c_1 = \{s_2\}, c_2 = \{s_4, s_5\}$ because $|C_{\mathsf{e}}(c_1)| <_s |C_{\mathsf{e}}(c_2)|$:

$$C_{\mathsf{e}}(c_1) = \{s_1, t_1\}$$
$$C_{\mathsf{e}}(c_2) = \{s_1, t_1, s_2, t_2, t_3, s_3, t_4\}$$

(3.12) applied to Σ in Figure 3.7 yields the three nets in Figure 3.8. For example, $(C_{\mathsf{e}}(c_2) - (C_{\mathsf{e}}(c_1) \cup c_1)) = \{s_3, t_2, t_3, t_4\}$ – Therefore:

$$\begin{aligned}
\Sigma'_2 &= \Sigma\,|_{(C_{\mathsf{e}}(c_2) - (C_{\mathsf{e}}(c_1) \cup c_1))} \\
&= (\{s_3\}, \{t_2, t_3, t_4\}, \{(t_2, s_3), (s_3, t_4)\}, \{(s_3, \emptyset), (t_2, \{b\}), \\
&\quad (t_3, \{d\}), (t_4, \{c\})\})
\end{aligned}$$

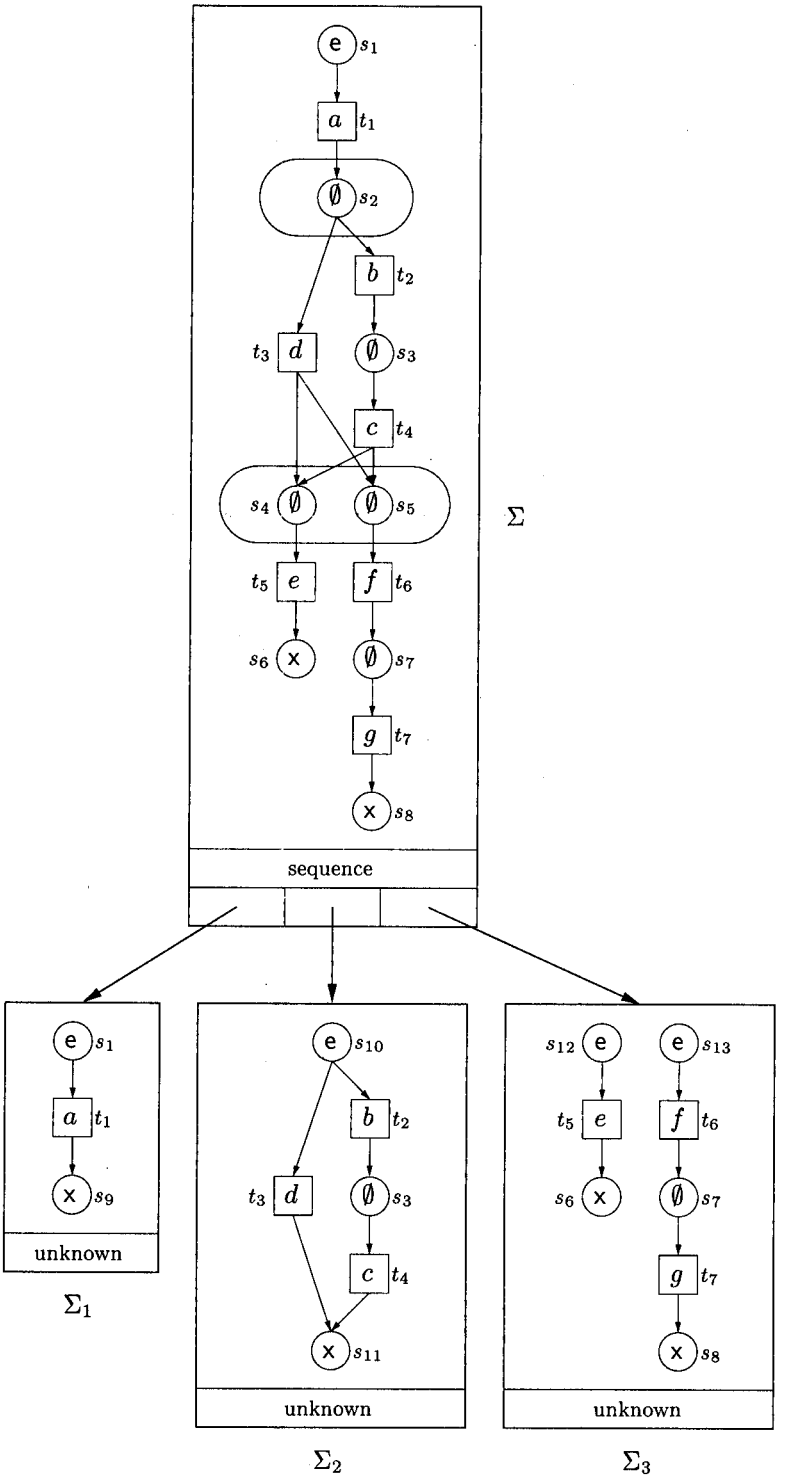


Figure 3.7: Sequential composition

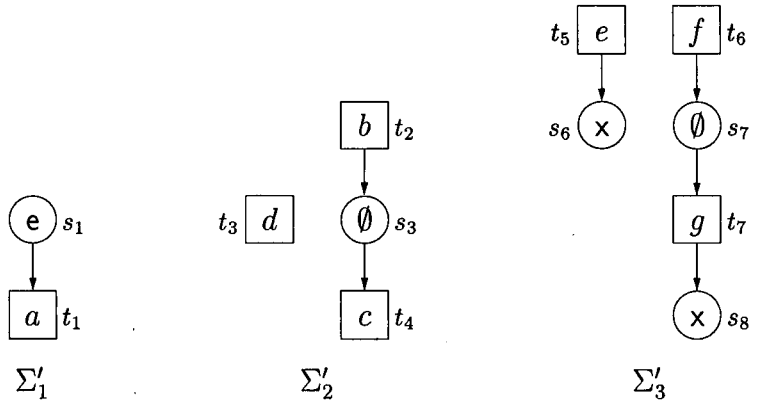


Figure 3.8: Partially decomposed sequence net

The interfaces obtained by applying the functions I_{e} and I_{x} (given by (3.13)) to the clusters c_1 and c_2, added to the nets in Figure 3.8 produce the subnets Σ_1, Σ_2 and Σ_3 shown in Figure 3.7. For example, the interfaces for Σ'_2 are produced using:

$$I_{\mathsf{e}}(c_1) = \{s_{10} = (\emptyset, \{t_2, t_3\})\}$$
$$I_{\mathsf{x}}(c_2) = \{s_{11} = (\{t_3, t_4\}, \emptyset)\}$$

Where s_{10} and s_{11} are introduced as shorthand for the place identifiers.

3.3.5 Iteration

Expression refinement

The expression E is always refined to:

$$E = [E_1 * E_2 * E_3]$$

Net decomposition

During the decomposition of the net, Σ, four auxiliary nets ($\Sigma_a - \Sigma_d$) are constructed. The first, Σ_a, is obtained by decomposing the entry and exit interfaces of Σ. The interface clusters given by S_{if} (in (3.15) below) are partially decomposed in Σ_a to give Σ_b. Proposition 15 shows that for any implementation, Σ, of a box expression which satisfies the preconditions of the iteration

synthesis rule, the decomposition of Σ is such that Σ_b consists of two disjoint components which are isomorphic to each other. Σ_c is taken to be one of these subnets. Σ_c contains all the information required to produce the decomposition into subnets Σ_1, Σ_2 and Σ_3.

The implementation of an iteration expression, $[E_1 * E_2 * E_3]$, involves two implementations of each of the expressions E_1, E_2 and E_3. The two implementations of E_1 are named Σ_{11} and Σ_{12}. The two implementations of E_2 and E_3 are named similarly.

Proposition 14 shows that the set of clusters, S_{if}, defined below will contain the two clusters of places arising from the interfaces ${\Sigma_{11}}^\bullet \otimes {}^\bullet\Sigma_{21} \otimes {\Sigma_{22}}^\bullet \otimes {}^\bullet\Sigma_{31}$ and ${\Sigma_{12}}^\bullet \otimes {}^\bullet\Sigma_{22} \otimes {\Sigma_{21}}^\bullet \otimes {}^\bullet\Sigma_{32}$.

$$\begin{aligned} S_{if} &= \{c \in C_{\mathsf{i}}(\Sigma) \mid (\exists t_e \in T_{\mathsf{e}}, \forall t \in T_{\mathsf{x}} : t_e \not\leftrightarrows_{N_{\mathsf{a}}-(S_{\mathsf{e}} \cup c)} t) \\ &\quad \wedge (\exists t_x \in T_{\mathsf{x}}, \forall t \in T_{\mathsf{e}} : t_x \not\leftrightarrows_{N_{\mathsf{a}}-(S_{\mathsf{x}} \cup c)} t)\} \end{aligned} \tag{3.15}$$

P_{e} (P_{x}) is defined to be a partition of the entry (exit) transitions of Σ into two sets corresponding to those entry transitions arising from Σ_{11} and Σ_{12} (exit transitions arising from Σ_{31} and Σ_{32}).

$$\begin{aligned} P_{\mathsf{e}} &= \{\{t \in T_{\mathsf{e}} \mid \forall t' \in T_{\mathsf{x}} : t \not\leftrightarrows_{N_{\mathsf{a}}-(S_{\mathsf{e}} \cup c)} t'\} \mid c \in S_{if}\} \\ P_{\mathsf{x}} &= \{\{t \in T_{\mathsf{x}} \mid \forall t' \in T_{\mathsf{e}} : t \not\leftrightarrows_{N_{\mathsf{a}}-(S_{\mathsf{x}} \cup c)} t'\} \mid c \in S_{if}\} \end{aligned}$$

The entry and exit interfaces of Σ can be decomposed in a fashion similar to that used for the choice synthesis rule, (3.6), using P_{e} and P_{x} as the set of equivalence classes determining the decomposition:

$$\begin{aligned} X_{\mathsf{e}} &= \{(\emptyset, s^\bullet \cap P) \mid P \in P_{\mathsf{e}}, s \in S_{\mathsf{e}}\} \\ X_{\mathsf{x}} &= \{({}^\bullet s \cap P, \emptyset) \mid P \in P_{\mathsf{x}}, s \in S_{\mathsf{x}}\} \end{aligned} \tag{3.16}$$

The first auxiliary net, Σ_a, is constructed by removing the entry and exit interfaces of Σ, and adding the interfaces defined by (3.16):

$$\Sigma_a = \Sigma \uplus (X_{\mathsf{e}}, e) \uplus (X_{\mathsf{x}}, x) \ominus (S_{\mathsf{e}} \cup S_{\mathsf{x}})$$

Attention now turns to the internal interfaces of Σ_a, which are given by S_{if}. Firstly, the set of all places comprising the internal interfaces is obtained: $S'_{if} = \{s \mid \exists C \in S_{if} : s \in C\}$. This set of places is decomposed into two new interfaces X_1 and X_2, obtained by taking just the incoming and outgoing arcs of the places in S'_{if}, respectively:

$$\begin{aligned} X_1 &= \{({}^\bullet s, \emptyset) \mid s \in S'_{if}\} \\ X_2 &= \{(\emptyset, s^\bullet) \mid s \in S'_{if}\} \end{aligned} \tag{3.17}$$

The net Σ_b is constructed by removing the original internal interface, and replacing it with the interfaces defined by X_1 and X_2. As shown in Proposition 15, Σ_b will consist of two disjoint nets, which are isomorphic to each other. There are two isomorphic nets in the partial decomposition because of the redundancy in the semantics for the iteration operator – one of the two nets is discarded, and the remaining one is named Σ_c.

$$\Sigma_b = \Sigma_a \uplus (X_1, \emptyset) \uplus (X_2, \emptyset) \ominus S'_{if}$$

The sets of entry and exit transitions of Σ_c, given by $T_{\mathsf{e}}(\Sigma_c)$ and $T_{\mathsf{x}}(\Sigma_c)$ respectively, are required later. The net Σ_c can be regarded as being isomorphic to a composition of implementations, $\Sigma_1, \Sigma_2, \Sigma_3$ of E_1, E_2 and E_3 in sequence, but with Σ_2 reversed – *i.e.* the entry interfaces of Σ_2 and Σ_3, and the exit interfaces of Σ_1 and Σ_2 are joined. Therefore, these two interfaces can be easily identified as they consist of internal places with no incoming or outgoing arcs respectively:

$$\begin{aligned} S_{\mathsf{i}_1} &= \{s \in S_{\mathsf{i}}(\Sigma_c) \mid s^\bullet = \emptyset\} \\ S_{\mathsf{i}_2} &= \{s \in S_{\mathsf{i}}(\Sigma_c) \mid {}^\bullet s = \emptyset\} \end{aligned} \tag{3.18}$$

The pre-transitions of S_{i_1}, and the post-transitions of S_{i_2} are partitioned into two sets according to whether there is a directed path from an entry place, or to an exit place respectively. The sets of transitions in P_{i_1} are such that every transition in the first set belongs to Σ_1, and those transitions in the second

set belong to Σ_2. Similarly with the partition P_{i_2}:

$$\begin{aligned} P_{i_1} &= \{\{t \in {}^\bullet S_{i_1} \mid \exists s \in S_{\mathsf{e}}(\Sigma_c) : s \vec{\sim} t\}, \{t \in {}^\bullet S_{i_1} \mid \nexists s \in S_{\mathsf{e}}(\Sigma_c) : s \vec{\sim} t\}\} \\ P_{i_2} &= \{\{t \in S^\bullet{}_{i_2} \mid \exists s \in S_{\mathsf{x}}(\Sigma_c) : t \vec{\sim} s\}, \{t \in S^\bullet{}_{i_2} \mid \nexists s \in S_{\mathsf{x}}(\Sigma_c) : t \vec{\sim} s\}\} \end{aligned} \quad (3.19)$$

These partitions are used to decompose the interfaces S_{i_1} (S_{i_2}) into the exit interfaces of Σ_1 and Σ_2 (entry interfaces of Σ_2 and Σ_3). The decomposition relies on the fact that no Petri box contains any duplicate places, as shown by Proposition 1 in Section 3.4.

$$\begin{aligned} X_{i_1} &= \{({}^\bullet s \cap P, \emptyset) \mid P \in P_{i_1}, s \in S_{i_1}\} \\ X_{i_2} &= \{(\emptyset, s^\bullet \cap P) \mid P \in P_{i_2}, s \in S_{i_2}\} \end{aligned} \quad (3.20)$$

The old internal interfaces (S_{i_1} and S_{i_2}) are removed from Σ_c, and the new interfaces given by (3.20) are added:

$$\Sigma_d = \Sigma_c \uplus (X_{i_1}, x) \uplus (X_{i_2}, e) \ominus (S_{i_1} \cup S_{i_2})$$

The three subnets, Σ_1, Σ_2 and Σ_3 can be identified in Σ_d as the connected component(s) containing the set of entry transitions, $T_{\mathsf{e}}(\Sigma_c)$, the connected component(s) containing neither the entry transitions nor the exit transitions, and the connected component(s) containing the set of exit transitions, $T_{\mathsf{x}}(\Sigma_c)$, respectively:

$$\begin{aligned} \Sigma_1 &= \Sigma_d \mid_{\mathcal{G}(S_{\mathsf{e}}(\Sigma_c))} \\ \Sigma_2 &= \Sigma_d \mid_{(N_{\mathsf{a}} - \mathcal{G}(S_{\mathsf{e}}(\Sigma_c) \cup S_{\mathsf{x}}(\Sigma_c)))} \\ \Sigma_3 &= \Sigma_d \mid_{\mathcal{G}(S_{\mathsf{x}}(\Sigma_c))} \end{aligned} \quad (3.21)$$

Optional checks

If the input net is the implementation of a box expression, then S_{if} is guaranteed to contain exactly two clusters of places. If $|S_{if}| \neq 2$ then the synthesis algorithm has detected that the input net is not an implementation of a box expression. The optional checks for the choice and sequence synthesis rules

can be reused to check that the net decomposition performed by the iteration synthesis is valid. If any of the checks fail, then it is known that the input net, Σ, is not the implementation of a box expression from the syntax in Table 3.1.

The decomposition of the entry and exit interfaces of Σ, given by X_{e} and X_{x} is similar to that used for the net decomposition in the choice synthesis rule. The optional checks for the choice synthesis rule can be reused here to ensure that the decompositions given by X_{e} and X_{x} are valid. The decomposition of the pair of clusters in S_{if} is the same as the decomposition of a cluster of places described in the sequence synthesis rule. The optional checks for the sequence synthesis rule can be used to check that the decomposition given by X_1 and X_2 (3.17) is valid.

If the input net is the implementation of a box expression, then Σ_b is guaranteed to consist of two isomorphic components, of which only one is required. In order to check that the input net really is the implementation of a box expression, both components of Σ_b must be checked. This means that the synthesis process needs to be applied to each component of Σ_b, and the resulting synthesised expressions checked for equivalence[1]. An algorithm for checking the equivalence of expressions is given in Section 3.5. The following considers the checks that need to be carried out on Σ_c, one of the components of Σ_b. The checks for the other component of Σ_b will be identical. The decomposition given by X_{i_1} and X_{i_2} (3.20) is the same as the decomposition of a cluster of places described in the sequence synthesis rule. The optional checks for the sequence synthesis rule can be reused to check that the decomposition is valid.

Example

Figure 3.9 shows an implementation of the expression:

$$[(a \,\square\, b) * (c; (d \parallel e)) * (f \parallel g)]$$

[1]The expressions should be equivalent, though not necessarily identical

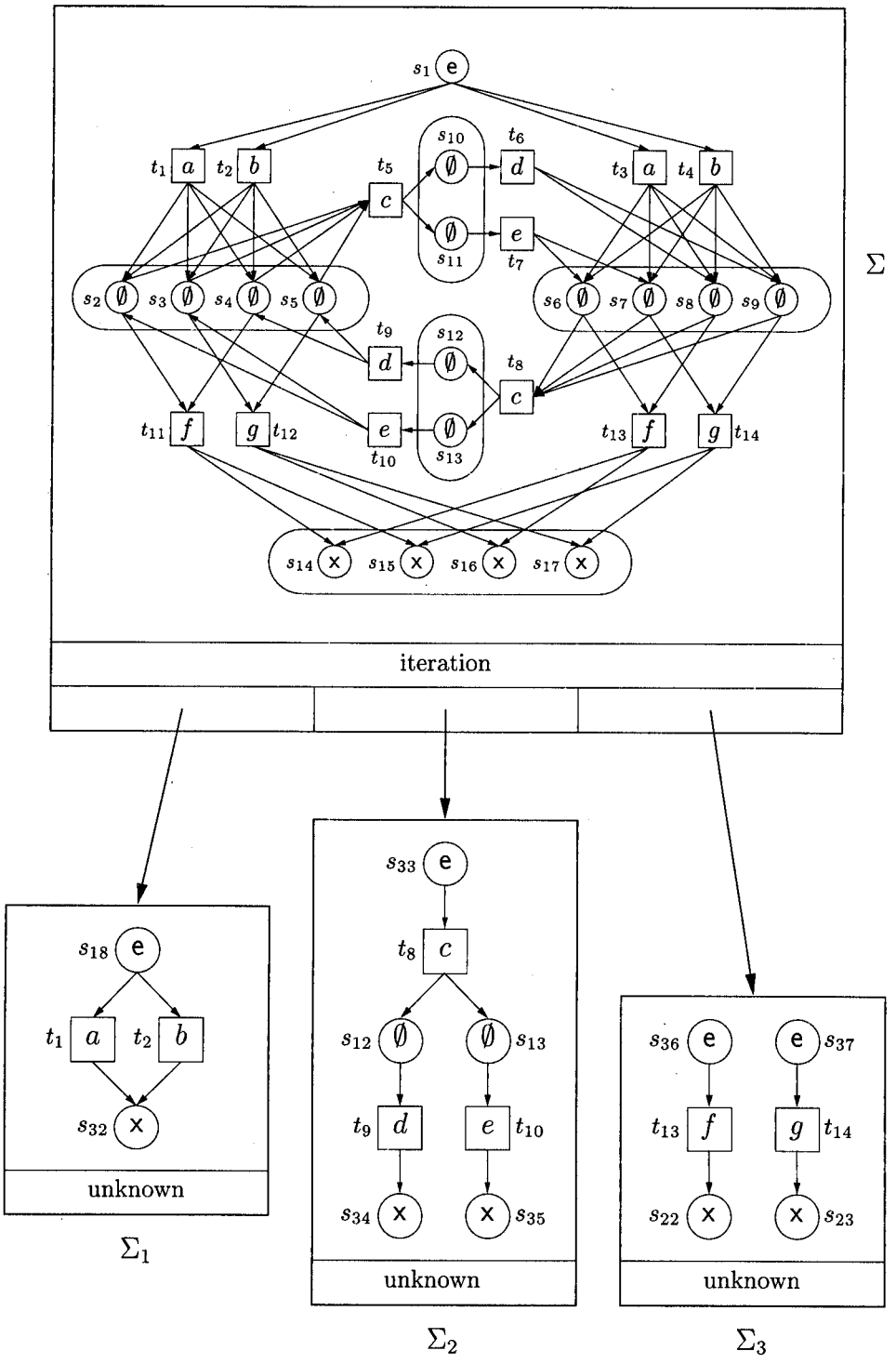


Figure 3.9: Iteration

This net contains more than one transition, is internally connected, and there is no cluster of places which, when removed, leaves no path between an entry and exit place. Therefore, the iteration synthesis rule is applicable. The set of internal clusters is given by

$$\mathcal{C}_{\mathsf{i}}(\Sigma) = \{\{s_2, s_3, s_4, s_5\}, \{s_{10}, s_{11}\}, \{s_6, s_7, s_8, s_9\}, \{s_{12}, s_{13}\}\}$$

as shown in Figure 3.9. (3.15) identifies the two internal clusters of interest as:

$$S_{if} = \{\{s_2, s_3, s_4, s_5\}, \{s_6, s_7, s_8, s_9\}\}$$

This provides the following partitioning of the entry and exit transitions:

$$\begin{aligned} P_{\mathsf{e}} &= \{\{t_1, t_2\}, \{t_3, t_4\}\} \\ P_{\mathsf{x}} &= \{\{t_{11}, t_{12}\}, \{t_{13}, t_{14}\}\} \end{aligned}$$

The net Σ_a constructed from Σ, by adding the interfaces defined by (3.16), is shown in Figure 3.10. Note that the identifiers s_{18} to s_{23} have been introduced as shorthand:

$$\begin{aligned} X_{\mathsf{e}} &= \{s_{18} = (\emptyset, \{t_1, t_2\}), s_{19} = (\emptyset, \{t_3, t_4\})\} \\ X_{\mathsf{x}} &= \{s_{20} = (\{t_{11}\}, \emptyset), s_{21} = (\{t_{12}, \emptyset\}), s_{22} = (\{t_{13}\}, \emptyset), s_{23} = (\{t_{14}\}, \emptyset)\} \end{aligned}$$

The set of places in the internal interfaces is $S'_{if} = \{s_2, ..., s_9\}$. By (3.17), the decomposition of these places is given by:

$$\begin{aligned} X_1 &= \{s_{24} = (\{t_1, t_2, t_{10}\}, \emptyset), s_{25} = (\{t_1, t_2, t_9\}, \emptyset), s_{26} = (\{t_3, t_4, t_6\}, \emptyset), \\ &\quad s_{27} = (\{t_3, t_4, t_7\}, \emptyset)\} \\ X_2 &= \{s_{28} = (\emptyset, \{t_5, t_{11}\}), s_{29} = (\emptyset, \{t_5, t_{12}\}), s_{30} = (\emptyset, \{t_8, t_{13}\}), \\ &\quad s_{31} = (\emptyset, \{t_8, t_{14}\})\} \end{aligned}$$

Once these interfaces have been added to the net in Figure 3.10, and one of the two isomorphic nets discarded, the net Σ_c is obtained. This net is shown

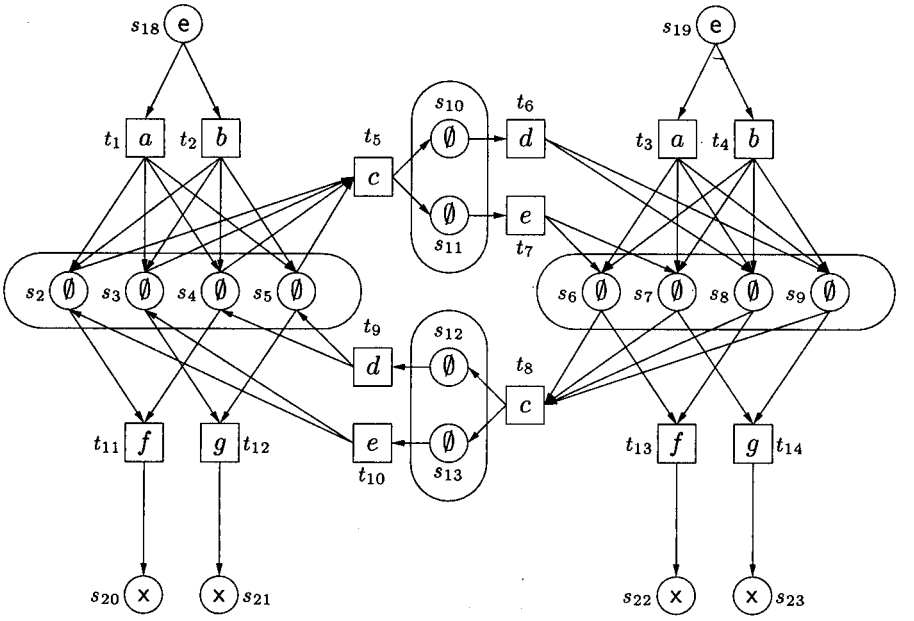


Figure 3.10: Auxiliary net, Σ_a

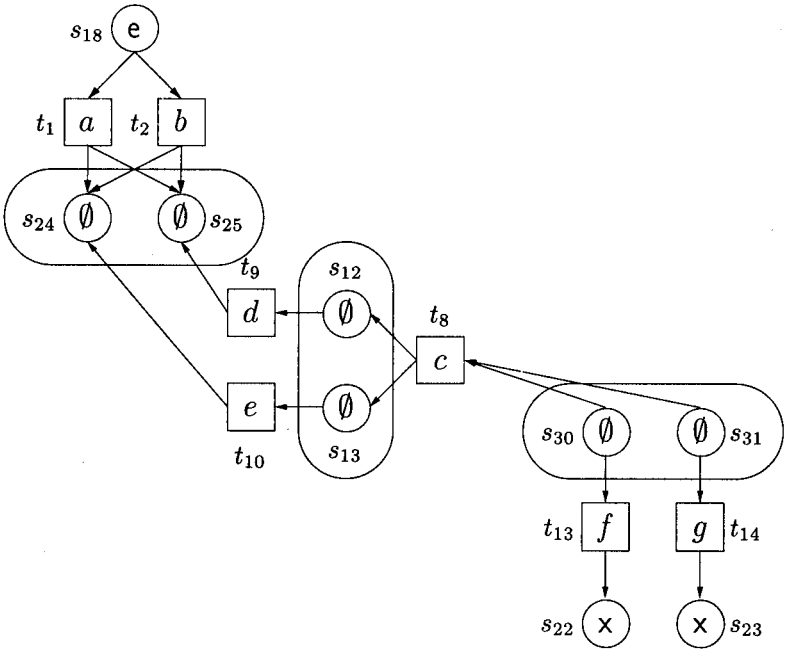


Figure 3.11: Auxiliary net, Σ_c

in Figure 3.11. The entry and exit places of Σ_c are $S_{\mathsf{e}}(\Sigma_c) = \{s_{18}\}$ and $S_{\mathsf{x}}(\Sigma_c) = \{s_{22}, s_{23}\}$ respectively.

The internal interfaces in Σ_c (those internal places with no incoming or outgoing arcs) are: $S_{\mathsf{i}_1} = \{s_{24}, s_{25}\}$ and $S_{\mathsf{i}_2} = \{s_{30}, s_{31}\}$. Therefore, by (3.19), the partitioning of the transitions is given by:

$$\begin{aligned} P_{\mathsf{i}_1} &= \{\{t_1, t_2\}, \{t_9, t_{10}\}\} \\ P_{\mathsf{i}_2} &= \{\{t_{13}, t_{14}\}, \{t_8\}\} \end{aligned}$$

Hence the decomposition of the interfaces described by (3.20) produces the disjoint union of the nets Σ_1, Σ_2 and Σ_3 in Figure 3.9. The interfaces are:

$$\begin{aligned} X_{\mathsf{i}_1} &= \{s_{32} = (\{t_1, t_2\}, \emptyset), s_{34} = (\{t_9\}, \emptyset), s_{35} = (\{t_{10}\}, \emptyset)\} \\ X_{\mathsf{i}_2} &= \{s_{33} = (\emptyset, \{t_8\}), s_{36} = (\emptyset, \{t_{13}\}), s_{37} = (\emptyset, \{t_{14}\})\} \end{aligned}$$

Where $s_{32}, ..., s_{37}$ are used as shorthand for the place identifiers. The set of places and transitions for Σ_1 is given by $\mathcal{G}(\{s_{18}\})$, and $\mathcal{G}(\{s_{22}, s_{23}\})$ for Σ_3. Therefore, the three subnets Σ_1, Σ_2 and Σ_3 can be identified, using (3.21).

3.4 Verification of the synthesis algorithm

This section verifies the correctness of the synthesis algorithm described in Sections 3.2 and 3.3.

Section 3.4.2 shows that the four properties of nets used to identify the synthesis rule to apply are correct. The following section proves that the decomposition performed by each synthesis rule is sound – *i.e.* if the decomposed nets are recombined according to the refined expression, a net isomorphic to the input net to the synthesis rule is obtained. In addition, it is shown that each decomposed subnet is the implementation of some expression from the language defined by Table 3.1. Finally, the correctness result of the algorithm is given in Section 3.4.4.

In the following proofs, let $\Sigma = (S, T, W, \lambda)$ be any implementation of a box expression, E, and, without loss of generality, assume for $1 \leq j \leq k$ (for

some k), $\Sigma_j = (S_j, T_j, W_j, \lambda_j)$ is the implementation of the subexpression E_j of E, such that every node in $N_{\mathsf{i}}(\Sigma_j)$ appears in Σ, and the connectivity between the nodes in $N_{\mathsf{i}}(\Sigma_j)$ is the same in both Σ_j and Σ – *i.e.*:

$$\forall n_1, n_2 \in N_{\mathsf{i}}(\Sigma_j) : W_j(n_1, n_2) = W(n_1, n_2) \wedge W_j(n_2, n_1) = W(n_2, n_1)$$

Similarly, for the subnets Σ_{jk}, for $1 \leq j \leq 3$, $1 \leq k \leq 2$ used in the iteration rule. For a set of places, S' in Σ, which corresponds to an application of the $\otimes$ operator, it is not necessary to decompose S' into sets of places S_1, S_2 such that $S_1 \otimes S_2 = S'$. Any decomposition such that $S_1 \otimes S_2 =_{iso} S'$, can be used.

3.4.1 Support proofs

Isolated and duplicated places

Proposition 1 *For every box expression, E, no implementation of E contains any isolated or duplicated places.*

Proof: By structural induction over the box expression syntax.

Base case: Any implementation of an atomic action, α, by definition, contains no isolated or duplicated places.

Induction step: The net semantics for each of the expressions $E = E_1 \parallel E_2$, $E = E_1; E_2$, $E = E_1 \,\square\, E_2$ and $E = [E_1 * E_2 * E_3]$ combines suitably disjoint implementations, Σ_i of the subexpressions, E_i. By the induction hypothesis no implementation of one of the subexpressions contains any isolated or duplicated places. A scheme for obtaining an implementation of E is described, and it is argued that such a scheme can not produce a net with either isolated or duplicated places. Furthermore, any net isomorphic to such an implementation will not contain any isolated or duplicated places.

1. The disjoint union of a collection of k disjoint subnets is taken:

$$\Sigma_a = \Sigma_1 \sqcup ... \sqcup \Sigma_k$$

By the definition of $\sqcup$, $\mathcal{I}(\Sigma_a) = \bigcup_{1 \leq i \leq k} \mathcal{I}(\Sigma_i)$. Therefore, by the induction hypothesis, $\mathcal{I}(\Sigma_a) = \emptyset$, and Σ_a does not contain any isolated places. For $1 \leq i, j \leq k$, such that $i \neq j$, no place originating from Σ_i duplicates a place originating from Σ_j because every place in Σ_i, and no place in Σ_j is connected to some transition in Σ_i. Therefore, Σ_a contains no duplicate places.

2. A set of places may be removed from the result of (1):

$$\Sigma_b = \Sigma_a \ominus X$$

Where X is the union of $2m$ (possibly zero) sets of places of the form ${}^\bullet\Sigma_i$ or $\Sigma_i{}^\bullet$, for some $1 \leq i \leq k$. Every place is only connected to transitions, not to other places. Therefore, Σ_b does not contain any isolated or duplicated places.

3. The sets of places removed in (2) are combined using zero or more applications of the $\otimes$ operator, and the resulting sets of places added to Σ_b. For example:

$$\Sigma_c = \Sigma_b \oplus (X_1 \otimes Y_1) \oplus ... \oplus (X_m \otimes Y_m)$$

In the case of iteration, the sets of places may be combined using nested applications of the $\otimes$ operator:

$$\Sigma_c = \Sigma_b \oplus ((X_1 \otimes Y_1) \otimes (X_2 \otimes Y_2)) \oplus ... \oplus ((X_{m-1} \otimes Y_{m-1}) \otimes (X_m \otimes Y_m))$$

Note that every set of places is unique, no place appears in more than one set, and for $1 \leq i \leq m$, the sets of places X_i and Y_i are from different subnets. By the induction hypothesis, the sets of places X_i and Y_i, for $1 \leq i \leq m$, do not contain any isolated or duplicated places. By the definition of the $\otimes$ operator, each place in

$X_i \otimes Y_i$ inherits the arcs of some place from X_i, and some place from Y_i, in such a way that no pair of places in $X_i \otimes Y_i$ duplicate each other. Hence, the result of a nested application of the $\otimes$ operator contains no isolated or duplicated places. Therefore, by definition of the $\oplus$ operator, Σ_c does not contain any isolated places. The sets of places X_i and Y_i are no longer present in Σ_b, and were either an entry or exit interface of one of the subnets. Hence each place in the sets resulting from an application of the $\otimes$ operator contains an arc to a transition, such that no place in Σ_b has a similar arc. Therefore Σ_c contains no duplicate places.

Hence, no implementation of a box expression contains any isolated or duplicated places. □

Place multiplication operator

Proposition 2 *For any application of the $\otimes$ operator, $S_1 \otimes S_2$, used in constructing an implementation of a box expression, every place in $S_1 \otimes S_2$ is connected to some transition in ${}^\bullet S_1 \cup S_1{}^\bullet$, and to some transition in ${}^\bullet S_2 \cup S_2{}^\bullet$, and every transition in ${}^\bullet S_1 \cup S_1{}^\bullet \cup {}^\bullet S_2 \cup S_2{}^\bullet$ is connected to some place in $S_1 \otimes S_2$. Furthermore, for any pair of places, s_1, s_2 in $S_1 \otimes S_2$, there is an undirected path between s_1 and s_2, and $s_1 \simeq_p s_2$* – i.e. *s_1 and s_2 belong to the same cluster.*

Note: With, for example, $S_1 = \Sigma_1{}^\bullet$, and $S_2 = {}^\bullet\Sigma_2$, then every place in $S_1 \otimes S_2$ is connected to some transition in ${}^\bullet S_1 = T_{\mathsf{X}}(\Sigma_1)$, and to some transition in $S_2{}^\bullet = T_{\mathsf{e}}(\Sigma_2)$, since $S_1{}^\bullet = \emptyset$ (exit places have no outgoing arcs), and ${}^\bullet S_2 = \emptyset$ (entry places have no incoming arcs). This property can be extended to all sets of places involved in nested applications of the $\otimes$ operator.

Proof: By Proposition 1, the sets of places, S_1 and S_2 do not contain any isolated places. By the definition of the $\otimes$ operator, for each pair of

places $s_1 \in S_1$ and $s_2 \in S_2$, there is a place in $S_1 \otimes S_2$ which inherits the arcs of s_1 and s_2. Hence every place in $S_1 \otimes S_2$ is connected to some transition in ${}^\bullet S_1 \cup S_1{}^\bullet$, and to some transition in ${}^\bullet S_2 \cup S_2{}^\bullet$, and every transition in ${}^\bullet S_1 \cup S_1{}^\bullet \cup {}^\bullet S_2 \cup S_2{}^\bullet$ is connected to some place in $S_1 \otimes S_2$. For any pair of places, x_1 and x_2 in $S_1 \otimes S_2$, the arcs of x_1 and x_2 must have been inherited from pairs of places p_1, p_2 and p_1', p_2' respectively, where $p_1, p_1' \in S_1$, and $p_2, p_2' \in S_2$. By the definition of the $\otimes$ operator, there exists a place x in $S_1 \otimes S_2$ which inherits its arcs from p_1 and p_2'. Therefore, there is an undirected path between x_1 and x_2. By definition of $\sim$ in Section 2.5.3, $x_1 \sim x$, and $x \sim x_2$. The cluster relation $\simeq_p$ is the transitive closure of $\sim$, therefore $x_1 \simeq_p x_2$. □

3.4.2 Verification of preconditions

Number of transitions (Precondition 1)

Proposition 3 *Any implementation of an atomic action expression contains exactly one transition, and every implementation of any other expression contains more than one transition.*

Proof: By structural induction over the box expression syntax.

Base case: The implementation of an atomic action, α, by definition, contains exactly one transition.

Induction step: By the induction hypothesis, the implementations of the subexpressions, E_i, in $E = E_1 \parallel E_2$, $E = E_1; E_2$, $E = E_1 \,\square\, E_2$ and $E = [E_1 * E_2 * E_3]$, contain at least one transition. By the compositional semantics of box expressions, and the definition of the $\sqcup$ operator, the cardinality of the set of transitions in an implementation of E is given by the sum of the number of transitions of the implementations of the subexpressions, E_i. Therefore, any implementation of E must contain

at least two transitions. Note that the $\ominus$ and $\otimes$ operators operate only on places, and so do not affect the number of transitions in a net. □

Connectedness properties (Preconditions 2 and 3)

Proposition 4 *For any box expression, E, and any implementation, Σ, of E, every node in $N_{\mathsf{i}}(\Sigma)$ is connected with respect to the relation, $\overset{\leftrightarrow}{\sim}_{N_{\mathsf{i}}}$, to some transition $t \in T_{\mathsf{e}}(\Sigma)$, and to some transition $t \in T_{\mathsf{x}}(\Sigma)$.*

Proof: By structural induction over the box expression syntax.

Base case: By definition, in any implementation of α, N_{i} consists of a single transition, t and $T_{\mathsf{e}} = T_{\mathsf{x}} = \{t\}$. Hence, the property holds for atomic actions.

Induction step: By the induction hypothesis, for any implementation, Σ_i of subexpression, E_i, every node in $N_{\mathsf{i}}(\Sigma_i)$ is connected to some transition in $T_{\mathsf{e}}(\Sigma_i)$, and some transition $T_{\mathsf{x}}(\Sigma_i)$. Let Σ be an implementation of E, constructed from disjoint implementations of the subexpressions.

- $E = E_1 \parallel E_2$, or, $E = E_1 \,\square\, E_2$: By the compositional semantics of the parallel and choice operators:

$$N_{\mathsf{i}}(\Sigma) = N_{\mathsf{i}}(\Sigma_1) \cup N_{\mathsf{i}}(\Sigma_2)$$
$$T_{\mathsf{e}}(\Sigma) = T_{\mathsf{e}}(\Sigma_1) \cup T_{\mathsf{e}}(\Sigma_2)$$
$$T_{\mathsf{x}}(\Sigma) = T_{\mathsf{x}}(\Sigma_1) \cup T_{\mathsf{x}}(\Sigma_2)$$

  Hence, every node in $N_{\mathsf{i}}(\Sigma)$ is connected to some transition in $T_{\mathsf{e}}(\Sigma)$ and some transition in $T_{\mathsf{x}}(\Sigma)$.

- $E = E_1; E_2$: By the compositional semantics of the sequence operator:

$$N_{\mathsf{i}}(\Sigma) = N_{\mathsf{i}}(\Sigma_1) \cup N_{\mathsf{i}}(\Sigma_2) \cup X$$
$$T_{\mathsf{e}}(\Sigma) = T_{\mathsf{e}}(\Sigma_1)$$
$$T_{\mathsf{x}}(\Sigma) = T_{\mathsf{x}}(\Sigma_2)$$

where $X = \Sigma_1{}^\bullet \otimes {}^\bullet\Sigma_2$. By Proposition 2, every transition in $T_{\mathsf{x}}(\Sigma_1) \cup T_{\mathsf{e}}(\Sigma_2)$ is connected to some place in X, and every place in X is connected to some transition in $T_{\mathsf{x}}(\Sigma_1)$, and to some transition in $T_{\mathsf{e}}(\Sigma_2)$. Therefore, every node in $N_{\mathsf{i}}(\Sigma)$ is connected to some transition in $T_{\mathsf{e}}(\Sigma)$ and some transition in $T_{\mathsf{x}}(\Sigma)$.

- $E = [E_1 * E_2 * E_3]$: By the compositional semantics of the iteration operator:

$$N_{\mathsf{i}}(\Sigma) = (\bigcup_{1 \le j \le 3, 1 \le k \le 2} N_{\mathsf{i}}(\Sigma_{jk})) \cup X_1 \cup X_2$$

$$T_{\mathsf{e}}(\Sigma) = T_{\mathsf{e}}(\Sigma_{11}) \cup T_{\mathsf{e}}(\Sigma_{12})$$

$$T_{\mathsf{x}}(\Sigma) = T_{\mathsf{x}}(\Sigma_{31}) \cup T_{\mathsf{x}}(\Sigma_{32})$$

where $X_1 = \Sigma_{11}{}^\bullet \otimes {}^\bullet\Sigma_{21} \otimes \Sigma_{22}{}^\bullet \otimes {}^\bullet\Sigma_{31}$, and $X_2 = \Sigma_{12}{}^\bullet \otimes {}^\bullet\Sigma_{22} \otimes \Sigma_{21}{}^\bullet \otimes {}^\bullet\Sigma_{32}$. By Proposition 2, every transition in ${}^\bullet X_1 \cup X_1{}^\bullet \cup {}^\bullet X_2 \cup X_2{}^\bullet$ is connected to some place in $X_1 \cup X_2$, and every place in X_1 (respectively X_2) is connected to some transition in $T_{\mathsf{x}}(\Sigma_{11})$ (respectively $T_{\mathsf{e}}(\Sigma_{31})$). Therefore, by the induction hypothesis, every node in $N_{\mathsf{i}}(\Sigma)$ is connected to some transition in $T_{\mathsf{e}}(\Sigma)$ and some transition in $T_{\mathsf{x}}(\Sigma)$.

Therefore, by the properties of isomorphism, for any implementation, Σ' of E, every node in $N_{\mathsf{i}}(\Sigma')$ is connected with respect to the relation, $\overset{\leftrightarrow}{\sim}_{N_{\mathsf{i}}}$ to some transition in $T_{\mathsf{e}}(\Sigma')$, and some transition in $T_{\mathsf{x}}(\Sigma')$. □

Corollary 1 *For any implementation, Σ, of a box expression, E, every node in Σ is connected with respect to the relation, $\overset{\leftrightarrow}{\sim}_{N_{\mathsf{a}}}$, to some transition $t \in T_{\mathsf{e}}(\Sigma)$, and to some transition $t \in T_{\mathsf{x}}(\Sigma)$.*

Proof: By Proposition 4, every node in $N_{\mathsf{i}}(\Sigma)$ is connected to some transition $t \in T_{\mathsf{e}}(\Sigma)$, and to some transition $t \in T_{\mathsf{x}}(\Sigma)$. By definition of T_{e} and T_{x}, and Proposition 1 every entry and exit place is connected to some node in $N_{\mathsf{i}}(\Sigma)$ (more particularly some node in $T_{\mathsf{e}}(\Sigma) \cup T_{\mathsf{x}}(\Sigma)$). Therefore,

every node in Σ is connected with respect to the relation, $\overset{\leftrightarrow}{\sim}_{N_a}$, to some transition $t \in T_e(\Sigma)$, and to some transition $t \in T_x(\Sigma)$. □

Proposition 5 *Let Σ be any implementation of a box expression, E. If the main connective of E is $\|$, then Σ is disjoint, otherwise Σ is connected. If the main connective of E is $\|$ or $\square$, then Σ is internally disjoint, otherwise Σ is internally connected.*

Proof: By Proposition 4 every internal node of Σ is connected to some transition in $T_e(\Sigma)$, and to some transition in $T_x(\Sigma)$.

- $E = \alpha$ – By definition, any implementation of α is both connected, and internally connected.
- $E = E_1 \| E_2$ – By Proposition 3, there are transitions, t_1 in Σ_1 and t_2 in Σ_2. By the compositional semantics of parallel composition, t_1 and t_2 are not connected to each other. Therefore, Σ is disjoint. When the entry and exit places of Σ are removed, the transitions corresponding to t_1 and t_2 are still present. Therefore, Σ is internally disjoint.
- $E = E_1 \square E_2$ – By Proposition 2, and the compositional semantics of choice, there is an undirected path between any pair of entry places in Σ, and by Proposition 1, every exit place of Σ is connected to some transition in $T_x(\Sigma)$. Therefore, by Proposition 4, Σ is connected. By the compositional semantics of parallel and choice operators, any implementation of $E_1 \square E_2$ with entry and exit places removed is isomorphic to an implementation of $E_1 \| E_2$ with entry and exit places removed. Hence, Σ is internally disjoint.
- $E = E_1; E_2$ – Let Σ' be an implementation of E, constructed from disjoint implementations, Σ_1, Σ_2 of E_1 and E_2. By the compositional semantics of the sequence operator, and Propositions 2 and 4, every internal node of Σ' is connected to some place in ${}^\bullet\Sigma_1 \otimes \Sigma_2{}^\bullet$, and

there is an undirected path between any pair of places in ${\Sigma_1}^\bullet \otimes {}^\bullet\Sigma_2$. Therefore, Σ' is internally connected. Hence, by the properties of isomorphism, Σ is internally connected. By Proposition 1, every entry and exit place of Σ is connected to some transition. Therefore Σ is connected.

- $E = [E_1 * E_2 * E_3]$ – Let Σ' be an implementation of E constructed from disjoint implementations of the subexpressions E_1, E_2 and E_3. By the compositional semantics of the iteration operator, and Propositions 2 and 4, every internal node of Σ' is connected to some place in $X_1 = {\Sigma_{11}}^\bullet \otimes {}^\bullet\Sigma_{21} \otimes {\Sigma_{22}}^\bullet \otimes {}^\bullet\Sigma_{31}$, or $X_2 = {\Sigma_{12}}^\bullet \otimes {}^\bullet\Sigma_{22} \otimes {\Sigma_{21}}^\bullet \otimes {}^\bullet\Sigma_{32}$. By Proposition 2, and Proposition 4, applied to either Σ_{21}, or Σ_{22}, every place in X_1 is connected to some place in X_2. By Proposition 2 there is an undirected path between any pair of places in X_2. Therefore, Σ' is internally connected. Hence by the properties of isomorphism, Σ is internally connected. By Proposition 1, every entry and exit place of Σ is connected to some transition. Therefore Σ is connected.

□

Cluster properties (Precondition 4)

Proposition 6 *Let Σ be an implementation of a box expression, E. The set of clusters of internal places of Σ is given by:*

$$\mathcal{C}_{\mathsf{i}}(\Sigma) =_{iso} \begin{cases} \emptyset & \text{if } E = \alpha \\ \mathcal{C}_{\mathsf{i}}(\Sigma_1) \cup \mathcal{C}_{\mathsf{i}}(\Sigma_2) & \text{if } E = E_1 \parallel E_2 \\ & \text{or } E = E_1 \,[]\, E_2 \\ \mathcal{C}_{\mathsf{i}}(\Sigma_1) \cup \mathcal{C}_{\mathsf{i}}(\Sigma_2) \cup \{{\Sigma_1}^\bullet \otimes {}^\bullet\Sigma_2\} & \text{if } E = E_1; E_2 \\ \bigcup_{1 \le j \le 3, 1 \le k \le 2} \mathcal{C}_{\mathsf{i}}(\Sigma_{jk}) \cup \{X_1, X_2\} & \text{if } E = [E_1 * E_2 * E_3] \end{cases}$$

where $X_1 = {\Sigma_{11}}^\bullet \otimes {}^\bullet\Sigma_{21} \otimes {\Sigma_{22}}^\bullet \otimes {}^\bullet\Sigma_{31}$, and $X_2 = {\Sigma_{12}}^\bullet \otimes {}^\bullet\Sigma_{22} \otimes {\Sigma_{21}}^\bullet \otimes {}^\bullet\Sigma_{32}$.

Proof: Any implementation of α, contains no internal places, and therefore, no clusters of internal places. By the definition of $\sqcup$:

$$\mathcal{C}_{\mathsf{i}}(\Sigma_1 \sqcup \Sigma_2) = \mathcal{C}_{\mathsf{i}}(\Sigma_1) \cup \mathcal{C}_{\mathsf{i}}(\Sigma_2)$$

Hence, by the compositional semantics of box expressions, there is a unique cluster of internal places in Σ corresponding to each cluster of internal places in the implementations of the subexpressions of E.

By the compositional semantics of box expressions, for any implementation Σ' of an expression, E', ${}^\bullet T_{\mathsf{e}}(\Sigma') = S_{\mathsf{e}}(\Sigma')$ and $T_{\mathsf{x}}(\Sigma')^\bullet = S_{\mathsf{x}}(\Sigma')$. Therefore, the places in the new clusters of internal places, constructed by the semantics of the sequence and iteration operators, are not related by $\simeq_p$ to places in any existing cluster of internal places. Hence, $\mathcal{C}_{\mathsf{i}}(\Sigma)$ gives the set of clusters of internal places of Σ. □

Proposition 7 *Let Σ be an implementation of a box expression, E. If the main connective of E is sequence, then there exists a cluster of internal places in Σ, which, when removed, leaves no undirected path between an entry place and an exit place in Σ, otherwise Σ contains no such cluster of places.*

Proof: The property is shown for an particular implementation, Σ', constructed from disjoint implementations of the subexpressions of E. By the properties of isomorphism, the proof also holds for an arbitrary implementation of E.

- $E = \alpha$ – By definition, any implementation of α contains no internal places.
- $E = E_1 \parallel E_2$ and $E = E_1 \,[]\, E_2$ – By Proposition 6, any candidate cluster of internal places, X, in Σ' originates from either Σ_1 or Σ_2. By the compositional semantics of the parallel and choice operators $T_{\mathsf{e}}(\Sigma') = T_{\mathsf{e}}(\Sigma_1) \cup T_{\mathsf{e}}(\Sigma_2)$, and $T_{\mathsf{x}}(\Sigma') = T_{\mathsf{x}}(\Sigma_1) \cup T_{\mathsf{x}}(\Sigma_2)$. Therefore, if X originates from Σ_1 (respectively Σ_2), there remains a path from an entry to exit place through Σ_2 (respectively Σ_1).

- $E = E_1; E_2$ – By Proposition 6, and the compositional semantics of sequence the interface between Σ_1 and Σ_2 formed by $X = \Sigma_1{}^\bullet \otimes {}^\bullet\Sigma_2$ is a candidate cluster. By the definition of $\sqcup$, the components Σ_1 and Σ_2 are disjoint in $\Sigma_1 \sqcup \Sigma_2$. Therefore, by definition of the $\ominus$ operator, there is no path between an entry place and an exit place in the net $\Sigma_a = \Sigma_1 \sqcup \Sigma_2 \ominus (\Sigma_1{}^\bullet \cup {}^\bullet\Sigma_2)$. By the compositional semantics of sequence, Σ_a is isomorphic to Σ' with the candidate cluster X removed.
- $E = [E_1 * E_2 * E_3]$ By the compositional semantics of iteration, and Proposition 6, the candidate clusters of internal places must originate entirely within one of the subnets, or from $X_1 = \Sigma_{11}{}^\bullet \otimes {}^\bullet\Sigma_{21} \otimes \Sigma_{22}{}^\bullet \otimes {}^\bullet\Sigma_{31}$, or $X_2 = \Sigma_{12}{}^\bullet \otimes {}^\bullet\Sigma_{22} \otimes \Sigma_{21}{}^\bullet \otimes {}^\bullet\Sigma_{32}$. By the compositional semantics of iteration:

$$\begin{aligned} T_{\mathsf{e}}(\Sigma') &= T_{\mathsf{e}}(\Sigma_{11}) \cup T_{\mathsf{e}}(\Sigma_{12}) \\ T_{\mathsf{x}}(\Sigma') &= T_{\mathsf{x}}(\Sigma_{31}) \cup T_{\mathsf{x}}(\Sigma_{32}) \end{aligned}$$

By Proposition 2, every transition in $T_{\mathsf{x}}(\Sigma_{11})$ ($T_{\mathsf{x}}(\Sigma_{12})$) is connected to each transition in $T_{\mathsf{e}}(\Sigma_{31})$ ($T_{\mathsf{e}}(\Sigma_{32})$) via a place in X_1 (X_2). Therefore, if X originates from Σ_{11}, Σ_{31} or X_1 (respectively Σ_{12}, Σ_{32} or X_2), there remains a path from an entry to exit place through Σ_{12}, X_2 and Σ_{32} (respectively Σ_{11}, X_1 and Σ_{31}). If X originates from Σ_{21}, or Σ_{22}, there remain paths from an entry to exit place through Σ_{12}, X_2 and Σ_{32}, and through Σ_{11}, X_1 and Σ_{31}.

□

Preconditions

Proposition 8 *For any implementation, Σ of a box expression, E from the syntax in Table 3.1, exactly one of the synthesis rules will be identified as applicable.*

Proof: By Propositions 3, 5, and 7, and Figure 3.3. □

3.4.3 Synthesis rule decomposition is sound

Atomic action

Proposition 9 *Let Σ be an implementation of a box expression. Whenever the atomic action synthesis rule is applicable to Σ, any implementation of the refined expression produced by the rule, is isomorphic to Σ.*

Proof: By Proposition 8, Σ contains exactly one transition, t. By Proposition 3, Σ is the implementation of an atomic action. By the compositional semantics of atomic actions, the label of t is the same as the atomic action expression. Therefore, the synthesis to $E = \lambda(t)$ is such that any implementation of E is isomorphic to Σ. □

Parallel composition

Proposition 10 *Let Σ be an implementation of a box expression. Whenever the parallel composition synthesis rule is applicable to Σ, the recomposition of the decomposed subnets, according to the refined expression produced by the rule, is isomorphic to Σ. Furthermore each of the decomposed subnets is an implementation of some box expression over the syntax in Table 3.1.*

Proof: By Proposition 8, Σ is an implementation of $E_1 \parallel ... \parallel E_k$, for some $k > 1$, where each E_i does not have $\parallel$ as the main connective. By the associativity of the parallel composition operator, shown in Lemma 5.15 in [6], no ambiguity is introduced by omitting the bracketing of subexpressions in $E_1 \parallel ... \parallel E_k$. Therefore, it is valid to decompose Σ directly into k subnets. By the compositional semantics of parallel:

$$\Sigma = \Sigma_1 \sqcup ... \sqcup \Sigma_k$$

By Proposition 5, Σ contains exactly k disjoint components (*i.e.* the number of connected components determines the value of k). By the def-

inition of the undirected connectedness relation, the equivalence classes of $\overset{\leftrightarrow}{\sim}_{N_a}$ correspond to the sets of nodes in each disjoint component of Σ. Hence Σ is decomposed into k subnets, corresponding to the unknown expressions $E_1, ..., E_k$, and the recomposition of these nets using the compositional semantics of parallel produces a net isomorphic to Σ. By the associativity and commutativity of the parallel composition operator (Lemma 5.15 in [6]), the ordering of the subexpressions in the refined expression is unimportant. □

Choice composition

Proposition 11 *For any choice expression, $E = E_1 \,[]\, E_2 \,[]\, ... \,[]\, E_k$, where for $1 \leq i \leq k$, the subexpression E_i does not have choice composition as its main connective, the relation $\sim_e$ is an equivalence relation over the set of entry transitions, $T_e(\Sigma)$, and the set of equivalence classes, P_{T_e}, partitions $T_e(\Sigma)$ into $T_e(\Sigma_1), T_e(\Sigma_2), ..., T_e(\Sigma_k)$.*

Proof: By the compositional semantics of choice:

$$\begin{aligned} \Sigma \;=_{iso}\; & \Sigma_1 \sqcup ... \sqcup \Sigma_k \\ & \oplus({}^\bullet\Sigma_1 \otimes ... \otimes {}^\bullet\Sigma_k, e) \oplus (\Sigma_1{}^\bullet \otimes ... \otimes \Sigma_k{}^\bullet, x) \\ & \ominus({}^\bullet\Sigma_1 \cup ... \cup {}^\bullet\Sigma_k \cup \Sigma_1{}^\bullet \cup ... \cup \Sigma_k{}^\bullet) \end{aligned} \tag{3.22}$$

Therefore, by definition of the $\oplus$ operator, the entry interface of Σ is isomorphic to ${}^\bullet\Sigma_1 \otimes {}^\bullet\Sigma_2 \otimes ... \otimes {}^\bullet\Sigma_k$. Hence, by definition of $\otimes$, and T_e:

$$T_e(\Sigma) = \bigcup_{1 \leq i \leq k} T_e(\Sigma_i)$$

Hence each $t \in T_e(\Sigma)$ belongs to exactly one $T_e(\Sigma_i)$, for some $1 \leq i \leq k$. For $t_1 \in T_e(\Sigma_i), t_2 \in T_e(\Sigma_j)$ such that $1 \leq i, j \leq k$ and $i \neq j$, it will be shown that $t_1 \not\sim_e t_2$. By definition of $\sim_e$:

$$t_1 \sim_e t_2 \Leftrightarrow t_1 \sim_{||}^* t_2 \vee t_1 \overset{\leftrightarrow}{\sim}_{N_1} t_2$$

Since t_1 and t_2 belong to different subnets, Σ_i and Σ_j, they cannot be internally connected in the net Σ. Therefore $t_1 \overleftrightarrow{\not\sim}_{N_i} t_2$. By definition of $\otimes$, $t_1 \not\sim^*_{\|} t_2$. Hence two entry transitions obtained from different subexpressions of the choice expression are not related by $\sim_{\mathsf{e}}$. The second part of the proof involves showing that for any $1 \leq i \leq k$, and $t_1, t_2 \in T_{\mathsf{e}}(\Sigma_i)$, then $t_1 \sim_{\mathsf{e}} t_2$. The main connective of E_i is not choice. It remains to check the other possibilities for the main connective:

- *E_i is an atomic action:* Σ_i contains a single transition. Therefore $t_1 \sim_{\mathsf{e}} t_2$ since $t_1 = t_2$ and both the relations $\overleftrightarrow{\sim}_{N_i}$ and $\sim^*_{\|}$ are reflexive.

- *Main connective of E_i is sequence or iteration:* By Proposition 5 $t_1 \overleftrightarrow{\sim}_{N_i} t_2$. Therefore, $t_1 \sim_{\mathsf{e}} t_2$.

- *Main connective of E_i is parallel:* E_i can be written as $F_1 \parallel F_2 \parallel \ldots \parallel F_m$, where $m > 1$, and the main connective of each F_j, for $1 \leq j \leq m$ is not $\|$. Let Σ_{F_j}, for $1 \leq j \leq m$ be disjoint implementations of F_j. By the compositional semantics of $\|$:

$$T_{\mathsf{e}}(\Sigma_i) = \bigcup_{1 \leq j \leq m} T_{\mathsf{e}}(\Sigma_{F_j})$$

  Hence each $t \in T_{\mathsf{e}}(\Sigma_i)$ belongs to exactly one $T_{\mathsf{e}}(\Sigma_{F_j})$, for some $1 \leq j \leq m$. If t_1 and t_2 arise from different subexpressions, F_g and F_h, then by the compositional semantics of $\|$, and definition of $\otimes$, $t_1 \sim_{\|} t_2$. Therefore $t_1 \sim^*_{\|} t_2$. If, however, t_1 and t_2 arose from the same subexpression, F_j, then there exists a transition $t_3 \in \Sigma_{F_g}$ such that $g \neq j$. By the compositional semantics of $\|$, and definition of $\otimes$, $t_1 \sim_{\|} t_3$ and $t_2 \sim_{\|} t_3$. Therefore $t_1 \sim^*_{\|} t_2$. In either case, $t_1 \sim_{\mathsf{e}} t_2$.

Hence $\sim_{\mathsf{e}}$ partitions $T_{\mathsf{e}}(\Sigma)$ into $P_{T_{\mathsf{e}}} = \{T_{\mathsf{e}}(\Sigma_1), T_{\mathsf{e}}(\Sigma_2), \ldots, T_{\mathsf{e}}(\Sigma_k)\}$, and $\sim_{\mathsf{e}}$ is an equivalence relation over $T_{\mathsf{e}}(\Sigma)$. □

Corollary 2 *For any choice expression, $E = E_1 \,[]\, E_2 \,[]\, ... \,[]\, E_k$, where for $1 \leq i \leq k$, the subexpression E_i does not have choice composition as its main connective, the set of equivalence classes P_{T_X} partitions $T_X(\Sigma)$ into $T_X(\Sigma_1), T_X(\Sigma_2)$, ..., $T_X(\Sigma_k)$.*

Proof: By Proposition 4, for $1 \leq i \leq k$, every transition $t \in T_X(\Sigma_i)$ is connected with respect to $\overleftrightarrow{\sim}_{N_I}$ to some transition $t' \in T_e(\Sigma_i)$. Furthermore, by the compositional semantics of the choice operator, there is no transition $t' \in T_e(\Sigma_j)$, such $i \neq j$ with $t \overleftrightarrow{\sim}_{N_I} t'$. Hence P_{T_X}, as defined by (3.4) is a set of equivalence classes which partitions $T_X(\Sigma)$ into $T_X(\Sigma_1), ..., T_X(\Sigma_k)$. □

Proposition 12 *Let Σ be an implementation of a box expression. Whenever the choice composition synthesis rule is applicable to Σ, the recomposition of the decomposed subnets, according to the refined expression produced by the rule, is isomorphic to Σ. Furthermore each of the decomposed subnets is the implementation of some box expression over the syntax in Table 3.1.*

Proof: By Proposition 8, Σ is an implementation of $E_1 \,[]\, ... \,[]\, E_k$ for some $k > 1$, where each E_i does not have $[]$ as the main connective. By the associativity and commutativity of the choice composition operator (Lemma 5.12 in [6]), no ambiguity is introduced by the omission of brackets, or the ordering of the subexpressions in $E_1 \,[]\, ... \,[]\, E_k$. Therefore, it is valid to decompose Σ directly into k subnets. By Proposition 11, and Corollary 2, P_{T_e} and P_{T_X} correspond to the partitioning of T_e and T_X into $T_e(\Sigma_1), ..., T_e(\Sigma_k)$ and $T_X(\Sigma_1), ..., T_X(\Sigma_k)$ respectively. Hence $|P_{T_e}|$ determines the value of k. Note that by the semantics of choice composition, and the definition of P_{T_X}, $|P_{T_X}| = |P_{T_e}|$.

By the compositional semantics of choice, and definition of the $\oplus$ operator, the entry and exit interfaces of Σ are isomorphic to ${}^\bullet\Sigma_1 \otimes {}^\bullet\Sigma_2 \otimes ... \otimes {}^\bullet\Sigma_k$ and ${\Sigma_1}^\bullet \otimes {\Sigma_2}^\bullet \otimes ... \otimes {\Sigma_k}^\bullet$ respectively. Therefore, by definition of the $\otimes$

operator, for each place $s \in {}^\bullet\Sigma$ ($s \in \Sigma^\bullet$), s has arcs to (from) some non empty set of transitions in $T_{\mathsf{e}}(\Sigma_i)$ ($T_{\mathsf{x}}(\Sigma_i)$) for $1 \leq i \leq k$. Again by the definition of $\otimes$ and $\oplus$, the arcs to (from) a particular set of transitions $T' \in T_{\mathsf{e}}(\Sigma_i)$ ($T' \in T_{\mathsf{x}}(\Sigma_i)$) were inherited from a single entry (exit) place of Σ_i. By Proposition 1, there are no isolated or duplicated places in the nets Σ_i for $1 \leq i \leq k$. Hence X_{e} and X_{x}, defined in (3.6) are the decomposition of ${}^\bullet\Sigma$ and $\Sigma^\bullet$ into the entry and exit interfaces of the subnets Σ_i for $1 \leq i \leq k$. Therefore (3.7) defines a net Σ_a which is isomorphic to the disjoint union of $\Sigma_1, \Sigma_2, ..., \Sigma_k$ by replacing the entry and exit interfaces of Σ with X_{e} and X_{x} respectively. By Corollary 1, for each $1 \leq i \leq k$, every node in Σ_i is connected to some transition $t \in T_{\mathsf{x}}(\Sigma_i)$. Therefore $\mathcal{G}(T_{\mathsf{x}}(\Sigma_i))$ is the set of nodes in Σ_i. By Corollary 2 $P_{T_{\mathsf{x}}}$ is the partition of $T_{\mathsf{x}}(\Sigma_a)$ into $T_{\mathsf{x}}(\Sigma_1), T_{\mathsf{x}}(\Sigma_2), ..., T_{\mathsf{x}}(\Sigma_k)$. Therefore, (3.8) decomposes Σ_a into the subnets $\Sigma_1, \Sigma_2, ..., \Sigma_k$, such that their choice composition is isomorphic to Σ, and for $1 \leq i \leq k$, Σ_i is the implementation of the subexpression E_i. Note that since the choice operator is associative and commutative, the ordering of the subexpressions in the refined expression is unimportant. □

Sequence

Proposition 13 *Let Σ be an implementation of a box expression. Whenever the sequential composition synthesis rule is applicable to Σ, the recomposition of the decomposed subnets, according to the refined expression produced by the rule, is isomorphic to Σ. Furthermore each of the decomposed subnets is an implementation of some box expression over the syntax in Table 3.1.*

Proof: By Proposition 8, Σ is the implementation of $E_1; ...; E_k$ for some $k > 1$, where each E_i does not have sequence as the main connective. By the associativity of the sequential composition operator (Lemma 5.9 in [6]), no ambiguity is introduced by omitting the bracketing of subexpressions in $E_1; ...; E_k$. Therefore, it is valid to decompose Σ directly into

k subnets. By the compositional semantics of sequence:

$$\begin{aligned}\Sigma \;=_{iso}\; & \Sigma_1 \sqcup \ldots \sqcup \Sigma_k \\ & \oplus(\Sigma_1^\bullet \otimes {}^\bullet\Sigma_2, \emptyset) \oplus \ldots \oplus (\Sigma_{k-1}^\bullet \otimes {}^\bullet\Sigma_k, \emptyset) \\ & \ominus(\Sigma_1^\bullet \cup \ldots \cup \Sigma_{k-1}^\bullet \cup {}^\bullet\Sigma_2 \cup \ldots \cup {}^\bullet\Sigma_k) \end{aligned} \tag{3.23}$$

By Proposition 7, Σ contains exactly $k-1$ internal clusters, each of which, when removed leaves no path between an entry and exit place. Note that the number of internal clusters determines the value of k. Hence, $S_;$, defined by (3.9) is the set of interface clusters which contains elements isomorphic to:

$$\Sigma_i^\bullet \otimes {}^\bullet\Sigma_{i+1} \text{ for } 1 \leq i \leq k-1$$

Removing any cluster of places $c \in S_;$ partitions Σ into two components, Σ_a and Σ_b, corresponding to the connected components containing the entry and exit places respectively. By (3.23):

$$\begin{aligned}\Sigma_a \;=_{iso}\; & \Sigma_1 \sqcup \ldots \sqcup \Sigma_i \\ & \oplus(\Sigma_1^\bullet \otimes {}^\bullet\Sigma_2, \emptyset) \oplus \ldots \oplus (\Sigma_{i-1}^\bullet \otimes {}^\bullet\Sigma_i, \emptyset) \\ & \ominus(\Sigma_1^\bullet \cup \ldots \cup \Sigma_i^\bullet \cup {}^\bullet\Sigma_2 \cup \ldots \cup {}^\bullet\Sigma_i) \\ \Sigma_b \;=_{iso}\; & \Sigma_{i+1} \sqcup \ldots \sqcup \Sigma_k \\ & \oplus(\Sigma_{i+1}^\bullet \otimes {}^\bullet\Sigma_{i+2}, \emptyset) \oplus \ldots \oplus (\Sigma_{k-1}^\bullet \otimes {}^\bullet\Sigma_k, \emptyset) \\ & \ominus(\Sigma_{i+1}^\bullet \cup \ldots \cup \Sigma_{k-1}^\bullet \cup {}^\bullet\Sigma_{i+2} \cup \ldots \cup {}^\bullet\Sigma_k) \end{aligned}$$

where c is isomorphic to $\Sigma_i^\bullet \otimes {}^\bullet\Sigma_{i+1}$, and by definition (3.11), $C_{\mathsf{e}}(c) = \Sigma_a$. By Proposition 3, each Σ_i contains at least one transition. Therefore, $<_s$ defines an ordering of the clusters in $S_;$, with $x_i =_{iso} \Sigma_i^\bullet \otimes {}^\bullet\Sigma_{i+1}$, for $1 \leq i \leq k-1$. Therefore, by (3.12):

$$\Sigma_i' = \begin{cases} \Sigma_1 \ominus \Sigma_1^\bullet & \text{if } i = 1 \\ \Sigma_i \ominus ({}^\bullet\Sigma_i \cup \Sigma_i^\bullet) & \text{if } 1 < i < k-1 \\ \Sigma_k \ominus {}^\bullet\Sigma_k & \text{if } i = k \end{cases}$$

For any net Σ_N, every place in $\Sigma_N{}^\bullet$ has no outgoing arcs, and every place in ${}^\bullet\Sigma_N$ has no incoming arcs. Therefore, by definition of $\otimes$ operator, for each place $s \in x_i$ (for each $x_i \in S_;$), the incoming arcs to s must have arisen from some place in $\Sigma_i{}^\bullet$, and the outgoing arcs from s must have arisen from some place in ${}^\bullet\Sigma_{i+1}$. Furthermore, the arcs of every place in $\Sigma_i{}^\bullet$ and ${}^\bullet\Sigma_{i+1}$ are represented at least once (possibly many times) in the places in x_i. By Proposition 1, there are no duplicate places in Σ_i. Therefore, the functions I_{e} and I_{x}, (3.13) can be used to decompose the clusters into $\Sigma_i{}^\bullet$ and ${}^\bullet\Sigma_{i+1}$, for $1 \leq i \leq k-1$. Hence the decomposition in (3.14) constructs implementations of the, as yet unknown, expressions $E_1, ..., E_k$, and the recomposition of these nets using the compositional semantics for the sequence operator produces a net isomorphic to Σ. □

Iteration

Let Σ, be any implementation of a box expression, such that Σ satisfies the preconditions of the the iteration rule. By Proposition 8, and the compositional semantics of iteration, there exists nets Σ_{jk} for $1 \leq j \leq 3, 1 \leq k \leq 2$ such that:

$$
\begin{aligned}
\Sigma \;=_{iso}\; & \Sigma_{11} \sqcup \Sigma_{12} \sqcup \Sigma_{21} \sqcup \Sigma_{22} \sqcup \Sigma_{31} \sqcup \Sigma_{32} \\
& \oplus({}^\bullet\Sigma_{11} \otimes {}^\bullet\Sigma_{12}, \mathsf{e}) \oplus (\Sigma_{11}{}^\bullet \otimes \Sigma_{12}{}^\bullet, \mathsf{x}) \\
& \ominus({}^\bullet\Sigma_{11} \cup {}^\bullet\Sigma_{12} \cup \Sigma_{31}{}^\bullet \cup \Sigma_{32}{}^\bullet) \\
& \oplus(\Sigma_{11}{}^\bullet \otimes {}^\bullet\Sigma_{21} \otimes \Sigma_{22}{}^\bullet \otimes {}^\bullet\Sigma_{31}, \emptyset) \\
& \oplus(\Sigma_{12}{}^\bullet \otimes {}^\bullet\Sigma_{22} \otimes \Sigma_{21}{}^\bullet \otimes {}^\bullet\Sigma_{32}, \emptyset) \\
& \ominus(\Sigma_{11}{}^\bullet \cup \Sigma_{12}{}^\bullet \cup {}^\bullet\Sigma_{21} \cup \Sigma_{21}{}^\bullet \cup {}^\bullet\Sigma_{22} \cup \Sigma_{22}{}^\bullet \cup {}^\bullet\Sigma_{31} \cup {}^\bullet\Sigma_{32})
\end{aligned}
$$

Proposition 14 *For any implementation, Σ, of a box expression, E, such that Σ satisfies the preconditions of the iteration rule. The set of internal clusters, S_{if}, defined in (3.15) gives the two clusters:*

$$S_{i1} \;=_{iso}\; (\Sigma_{11}{}^\bullet \otimes {}^\bullet\Sigma_{21} \otimes \Sigma_{22}{}^\bullet \otimes {}^\bullet\Sigma_{31}, \emptyset)$$

$$S_{i2} =_{iso} (\Sigma_{12}^{\bullet} \otimes {}^{\bullet}\Sigma_{22} \otimes \Sigma_{21}^{\bullet} \otimes {}^{\bullet}\Sigma_{32}, \emptyset)$$

Proof: By Proposition 6, the set of clusters of internal places, $\mathcal{C}_{\mathsf{i}}(\Sigma)$ is given by:

$$\mathcal{C}_{\mathsf{i}}(\Sigma) =_{iso} \mathcal{C}_{\mathsf{i}}(\Sigma_{11}) \cup \mathcal{C}_{\mathsf{i}}(\Sigma_{12}) \cup \mathcal{C}_{\mathsf{i}}(\Sigma_{21}) \cup \mathcal{C}_{\mathsf{i}}(\Sigma_{22}) \cup \mathcal{C}_{\mathsf{i}}(\Sigma_{31}) \cup \mathcal{C}_{\mathsf{i}}(\Sigma_{32}) \cup S_{\mathsf{i}1} \cup S_{\mathsf{i}2}$$

The following points are used in the proof:

1. By the compositional semantics of iteration:

$$\begin{aligned} T_{\mathsf{e}}(\Sigma) &= T_{\mathsf{e}}(\Sigma_{11}) \cup T_{\mathsf{e}}(\Sigma_{12}) \\ T_{\mathsf{x}}(\Sigma) &= T_{\mathsf{x}}(\Sigma_{31}) \cup T_{\mathsf{x}}(\Sigma_{32}) \end{aligned}$$

2. By Proposition 4, for each subnet, $\Sigma' \in \{\Sigma_{11}, \Sigma_{12}, \Sigma_{21}, \Sigma_{22}, \Sigma_{31}, \Sigma_{32}\}$

$$\forall t_1 \in T_{\mathsf{e}}(\Sigma'), t_2 \in T_{\mathsf{x}}(\Sigma') : t_1 \overset{\leftrightarrow}{\sim}_{N_{\mathsf{i}}(\Sigma')} t_2$$

3. By definition of the $\otimes$ and $\oplus$ operators, for any pair of transitions $t_1 \in T_{\mathsf{x}}(\Sigma_{11})$, and $t_2 \in T_{\mathsf{e}}(\Sigma_{21})$, or $t_2 \in T_{\mathsf{e}}(\Sigma_{31})$ then there exists a place in $S_{\mathsf{i}1}$ which has arcs connected to both t_1 and t_2. Similarly, for any pair of transitions $t_1 \in T_{\mathsf{x}}(\Sigma_{12})$, and $t_2 \in T_{\mathsf{x}}(\Sigma_{21})$, or $t_2 \in T_{\mathsf{e}}(\Sigma_{32})$ then there exists a place in $S_{\mathsf{i}2}$ which has arcs connected to both t_1 and t_2.

Firstly, it is shown that an internal cluster, c, arising from one of the subnets of Σ cannot satisfy the conditions for inclusion in S_{if}.

- c **is from** Σ_{11} **or** Σ_{12} **:** By 1, 2, and 3:

$$\forall t \in T_{\mathsf{x}}(\Sigma) \exists t_1 \in T_{\mathsf{x}}(\Sigma_{11}), t_2 \in T_{\mathsf{x}}(\Sigma_{12} : t \overset{\leftrightarrow}{\sim}_{N'} t_1 \wedge t \overset{\leftrightarrow}{\sim}_{N'} t_2$$

  where $N' = N_{\mathsf{i}}(\Sigma_{31}) \cup N_{\mathsf{i}}(\Sigma_{32}) \cup N_{\mathsf{i}}(\Sigma_{21}) \cup S_{\mathsf{i}1} \cup S_{\mathsf{i}2} \cup T_{\mathsf{x}}(\Sigma_{11}) \cup T_{\mathsf{x}}(\Sigma_{12})$. If c is from Σ_{11} or Σ_{12}, then $N' \subset N_{\mathsf{a}} - (S_{\mathsf{x}} \cup c)$. Hence, by 2, if c is from Σ_{11} then:

$$\forall t \in T_{\mathsf{x}}(\Sigma) \exists t' \in T_{\mathsf{e}}(\Sigma_{12}) : t \overset{\leftrightarrow}{\sim}_{N_{\mathsf{a}} - (S_{\mathsf{x}} \cup c)} t'$$

and if c is from Σ_{12} then:

$$\forall t \in T_{\mathsf{x}}(\Sigma) \exists t' \in T_{\mathsf{e}}(\Sigma_{11}) : t \overset{\leftrightarrow}{\sim}_{N_{\mathsf{a}}-(S_{\mathsf{x}} \cup c)} t'$$

Therefore, by 1 and (3.15), c cannot belong to S_{if}.

- c **is from** Σ_{21} **or** Σ_{22} **:** By 1, 2, and 3:

$$\forall t \in T_{\mathsf{e}}(\Sigma) \exists t' \in T_{\mathsf{x}}(\Sigma) : t \overset{\leftrightarrow}{\sim}_{N'} t'$$

where $N' = N_{\mathsf{i}}(\Sigma_{11}) \cup N_{\mathsf{i}}(\Sigma_{12}) \cup S_{\mathsf{i}_1} \cup S_{\mathsf{i}_2} \cup N_{\mathsf{i}}(\Sigma_{31}) \cup N_{\mathsf{i}}(\Sigma_{32})$. If c is in Σ_{21} or Σ_{22}, then $N' \subset N_{\mathsf{a}} - (S_{\mathsf{e}} \cup c)$. Hence:

$$\forall t \in T_{\mathsf{e}}(\Sigma) \exists t' \in T_{\mathsf{x}}(\Sigma) : t \overset{\leftrightarrow}{\sim}_{N_{\mathsf{a}}-(S_{\mathsf{e}} \cup c)} t'$$

Therefore, by (3.15), c cannot belong to S_{if}.

- c **is from** Σ_{31} **or** Σ_{32} **:** Symmetric to the case where c is from Σ_{11} or Σ_{12}.

Secondly, the two clusters, S_{i_1} and S_{i_2} are shown to satisfy the conditions for inclusion in S_{if}. By the compositional semantics of iteration, removing S_{i_1} and S_{e} (S_{i_1} and S_{x}) from Σ leaves a net in which the nodes of $N_{\mathsf{i}}(\Sigma_{11})$ ($N_{\mathsf{i}}(\Sigma_{31})$) are disjoint from the other subnets. Hence, by 1, no transition in $T_{\mathsf{e}}(\Sigma_{11})$ (in $T_{\mathsf{x}}(\Sigma_{31})$) is connected to a transition in $T_{\mathsf{x}}(\Sigma)$ (in $T_{\mathsf{e}}(\Sigma)$). Therefore, by (3.15), S_{i_1} belongs to S_{if}. A similar argument can be applied to S_{i_2} with subnets Σ_{12} and Σ_{32}. Hence $S_{if} = \{S_{\mathsf{i}_1}, S_{\mathsf{i}_2}\}$. □

Proposition 15 *Let Σ be an implementation of a box expression. Whenever the iteration synthesis rule is applicable to Σ, the recomposition of the decomposed subnets, according to the refined expression produced by the rule, is isomorphic to Σ. Furthermore each of the decomposed subnets is isomorphic to the natural implementation of some box expression over the syntax in Table 3.1.*

Proof: By Proposition 14, $S_{if} = \{S_{i1}, S_{i2}\}$. By the compositional semantics of iteration, removing S_e and S_{i1} (S_e and S_{i2}) from Σ leaves a net in which the nodes of $N_i(\Sigma_{11})$ ($N_i(\Sigma_{12})$) are disjoint from the other subnets. However, the nodes of $N_i(\Sigma_{12})$ ($N_i(\Sigma_{11})$) remain connected to the exit places by S_{i2} (S_{i1}). Hence:

$$P_e = \{T_e(\Sigma_{11}), T_e(\Sigma_{12})\}$$

A similar argument applied to the removal of $S_x(\Sigma)$, and either S_{i1} or S_{i2} shows:

$$P_x = \{T_x(\Sigma_{31}), T_x(\Sigma_{32})\}$$

By the semantics of iteration, $T_e(\Sigma) = T_e(\Sigma_{11}) \cup T_e(\Sigma_{12})$ and $T_x(\Sigma) = T_x(\Sigma_{31}) \cup T_x(\Sigma_{32})$. Therefore, by Proposition 1, X_e and X_x, defined in (3.16) are the decomposition of ${}^\bullet\Sigma$ and $\Sigma^\bullet$ into the entry interfaces of Σ_{11} and Σ_{12}, and the exit interfaces of Σ_{31} and Σ_{32}. Hence:

$$\begin{aligned}
\Sigma_a \;=_{iso}\; & \Sigma_{11} \sqcup \Sigma_{12} \sqcup \Sigma_{21} \sqcup \Sigma_{22} \sqcup \Sigma_{31} \sqcup \Sigma_{32} \\
& \oplus(\Sigma_{11}{}^\bullet \otimes {}^\bullet\Sigma_{21} \otimes \Sigma_{22}{}^\bullet \otimes {}^\bullet\Sigma_{31}, \emptyset) \\
& \oplus(\Sigma_{12}{}^\bullet \otimes {}^\bullet\Sigma_{22} \otimes \Sigma_{21}{}^\bullet \otimes {}^\bullet\Sigma_{32}, \emptyset) \\
& \ominus(\Sigma_{11}{}^\bullet \cup \Sigma_{12}{}^\bullet \cup {}^\bullet\Sigma_{21} \cup \Sigma_{21}{}^\bullet \cup {}^\bullet\Sigma_{22} \cup \Sigma_{22}{}^\bullet \cup {}^\bullet\Sigma_{31} \cup {}^\bullet\Sigma_{32})
\end{aligned}$$

By Proposition 14, and (3.17), X_1 corresponds to the set of places $(\Sigma_{11}{}^\bullet \otimes \Sigma_{22}{}^\bullet) \cup (\Sigma_{12}{}^\bullet \otimes \Sigma_{21}{}^\bullet)$ and X_2 corresponds to $({}^\bullet\Sigma_{31} \otimes {}^\bullet\Sigma_{21}) \cup ({}^\bullet\Sigma_{32} \otimes {}^\bullet\Sigma_{22})$. Therefore:

$$\begin{aligned}
\Sigma_b \;=_{iso}\; & \Sigma_{11} \sqcup \Sigma_{22} \sqcup \Sigma_{32} \\
& \oplus(\Sigma_{11}{}^\bullet \otimes \Sigma_{22}{}^\bullet, \emptyset) \oplus ({}^\bullet\Sigma_{22} \otimes {}^\bullet\Sigma_{32}, \emptyset) \\
& \ominus(\Sigma_{11}{}^\bullet \cup {}^\bullet\Sigma_{22} \cup \Sigma_{22}{}^\bullet \cup {}^\bullet\Sigma_{32}) \\
\sqcup\; & \Sigma_{12} \sqcup \Sigma_{21} \sqcup \Sigma_{31} \\
& \oplus(\Sigma_{12}{}^\bullet \otimes \Sigma_{21}{}^\bullet, \emptyset) \oplus ({}^\bullet\Sigma_{21} \otimes {}^\bullet\Sigma_{31}, \emptyset) \\
& \ominus(\Sigma_{12}{}^\bullet \cup {}^\bullet\Sigma_{21} \cup \Sigma_{21}{}^\bullet \cup {}^\bullet\Sigma_{31})
\end{aligned}$$

Since, for $1 \leq i \leq 3$, Σ_{i1} and Σ_{i2} are isomorphic implementations of E_i, then Σ_b consists of two isomorphic nets. Hence, Σ_c which is defined to be one of the two nets can be given by:

$$\begin{aligned}\Sigma_c \;=_{iso}\; & \Sigma_1 \sqcup \Sigma_2 \sqcup \Sigma_3 \\ & \oplus(\Sigma_1{}^\bullet \otimes \Sigma_2{}^\bullet, \emptyset) \oplus ({}^\bullet\Sigma_2 \otimes {}^\bullet\Sigma_3, \emptyset) \\ & \ominus(\Sigma_1{}^\bullet \cup {}^\bullet\Sigma_2 \cup \Sigma_2{}^\bullet \cup {}^\bullet\Sigma_3)\end{aligned}$$

By (3.18), S_1 corresponds to the set of places $\Sigma_1{}^\bullet \otimes \Sigma_2{}^\bullet$ and S_2 corresponds to ${}^\bullet\Sigma_2 \otimes {}^\bullet\Sigma_3$. By the compositional semantics of Σ_c, $S_{\mathsf{e}}(\Sigma_c) = S_{\mathsf{e}}(\Sigma_1)$ and $S_{\mathsf{x}}(\Sigma_c) = S_{\mathsf{x}}(\Sigma_3)$. Therefore:

$$\begin{aligned}P_{\mathsf{i}_1} &= \{T_{\mathsf{x}}(\Sigma_1), T_{\mathsf{x}}(\Sigma_2)\} \\ P_{\mathsf{i}_2} &= \{T_{\mathsf{e}}(\Sigma_3), T_{\mathsf{e}}(\Sigma_2)\}\end{aligned}$$

Therefore, by Proposition 1, X_{i_1} and X_{i_2}, defined in (3.20) are the decomposition of S_1 and S_2 into the entry interfaces of Σ_2 and Σ_3, and the exit interfaces of Σ_1 and Σ_2. Hence:

$$\Sigma_d = \Sigma_1 \sqcup \Sigma_2 \sqcup \Sigma_3$$

By the compositional semantics of Σ_c, $T_{\mathsf{e}}(\Sigma_c) = T_{\mathsf{e}}(\Sigma_1)$ and $T_{\mathsf{x}}(\Sigma_c) = T_{\mathsf{x}}(\Sigma_3)$. Therefore, by Proposition 4, the definition of T_{e} and T_{x}, and the compositional semantics of Σ_d:

$$\begin{aligned}\Sigma_d \mid_{\mathcal{G}(S_{\mathsf{e}}(\Sigma_c))} &= \Sigma_1 \\ \Sigma_d \mid_{\mathcal{G}(S_{\mathsf{x}}(\Sigma_c))} &= \Sigma_3 \\ \Sigma_d \mid_{N_{\mathsf{a}} - \mathcal{G}(S_{\mathsf{e}}(\Sigma_c) \cup S_{\mathsf{x}}(\Sigma_c))} &= \Sigma_2\end{aligned}$$

The nets Σ_1, Σ_2 and Σ_3 are implementations of the, as yet unknown, expressions E_1,E_2,and E_3, and the recomposition of these nets using the compositional semantics for iteration produces a net isomorphic to Σ. □

3.4.4 Correctness of the algorithm

Theorem 1 *For any implementation, Σ, of a box expression, E, from the syntax in Table 3.1, given the input net, Σ, the synthesis algorithm terminates with an output expression, E', such that any implementation of E' is isomorphic to Σ.*

Proof: By induction on the number of transitions in the net.

Base case (number of transitions=1): By Proposition 3, the synthesis rule applied is the atomic action rule. By Proposition 9, the net is synthesised to an expression, E', such that any implementation of E' is isomorphic to Σ.

Induction step (number of transitions=n): By Proposition 8, one of the synthesis rules will be applied, producing a refined expression, R. By Propositions 10, 12, 13 and 15, Σ will be decomposed into a finite collection of subnets, $\Sigma_1, ..., \Sigma_k$, for some $k > 1$, and the sum of the number of transitions in $\Sigma_1, ..., \Sigma_k$ is equal to n (the number of transitions in Σ). The refined expression R, will contain $E_1, ..., E_k$, references to the, as yet, unknown expressions, of which, the nets $\Sigma_1, ..., \Sigma_k$ are implementations. By Propositions 10, 12, 13 and 15, $E_1, ..., E_k$ can be represented using the box expression syntax in Table 3.1. Hence, by Proposition 3, each of the nets $\Sigma_1, ..., \Sigma_k$ must contain at least one transition, and therefore less than n transitions. Therefore, by the induction hypothesis, the synthesis algorithm, given each of the nets $\Sigma_1, ..., \Sigma_k$ as input will terminate with output expressions $E'_1, ..., E'_k$, such that for $1 \leq i \leq k$, the implementation of E'_i is isomorphic to Σ_i. The output expression, E' is obtained by replacing the references $E_1, ..., E_k$ in R, by the expressions $E'_1, ..., E'_k$. Hence the synthesis algorithm terminates on input Σ. By the Propositions in Section 3.4.3, any implementation of E' will be isomorphic to the input net, Σ. □

3.5 Related problems

In this section, the time complexity of the synthesis algorithm is shown to be polynomial. This result is used to show that the solutions, presented for those related problems which use the synthesis algorithm, are efficient. The points of non-determinism in the algorithm, highlighted by the analysis in Section 3.4 are discussed. This leads to a definition of a canonical form for box expressions from the language generated by the syntax in Table 3.1, and an algorithm for CANONICAL BOX EXPRESSION SYNTHESIS. Solutions to PETRI BOX ISOMORPHISM and BOX EXPRESSION ISOMORPHISM are presented in Section 3.5.4. A set of axioms is introduced, and shown to be complete. These axioms are used in the proofs generated by BOX EXPRESSION ISOMORPHISM PROOF. The algorithm for BOX EXPRESSION ISOMORPHISM PROOF is presented in Section 3.5.6. Finally, examples of the use of each of the algorithms are given in Section 3.5.7.

3.5.1 Time complexity

The analysis of the time complexity in this section is based on the size of the input net, $\Sigma = (S, T, W, \lambda)$. For simplicity, it will be assumed that the size of each transition label is bounded by some constant. Let $n = |S| + |T|$, and a be the number of nodes and arcs respectively, in Σ. The number of arcs, a, is bounded by $|S| \cdot |T| < n^2$ because Σ is bipartite, and there is at most one arc between any pair of nodes. Hence, the time complexity of BOX EXPRESSION SYNTHESIS will be given in terms of n.

The time complexity of checking the four structural properties in the ANALYSE function is given in Table 3.3. In the worst case, all four properties must be checked. Therefore, ANALYSE has time complexity $O(n^3)$. The properties Pr_2, Pr_3 and Pr_4 can be checked using variants of the depth-first search algorithm, which has time complexity $O(n+a)$. Property Pr_4 requires at most $|S|$ applications of depth-first search.

Property	Time complexity
Property Pr_1	$O(1)$
Property Pr_2	$O(n^2)$
Property Pr_3	$O(n^2)$
Property Pr_4	$O(n^3)$

Table 3.3: Time complexity of checking properties

The five synthesis rules have the time complexity shown in Table 3.4. The net decomposition for PARALLEL uses an extension of the algorithm for checking property Pr_2. In CHOICE, computing the equivalence classes of the relation $\sim_{\mathsf{e}}$, involves looking at every pair of arcs of every pair of entry transitions. The number of such pairs of arcs is bounded by $|T|^2 \cdot |S|^2 < n^4$. The time complexity of SEQUENCE is dominated by the time taken to find the set of clusters of internal places (an extension of the algorithm for checking property Pr_4). A similar procedure is used in ITERATION to identify the set of clusters, S_{if}.

Synthesis rule	Time complexity
ATOMIC	$O(1)$
PARALLEL	$O(n^2)$
CHOICE	$O(n^4)$
SEQUENCE	$O(n^3)$
ITERATION	$O(n^3)$

Table 3.4: Time complexity of the synthesis rules

Each decomposition has time complexity $O(n^4)$ where n is the number of nodes in the net at the node being decomposed, and each decomposed net has fewer nodes than the original net. During the decomposition of Σ, transitions are either discarded when the iteration rule is applied, or are synthesised to an atomic action. Therefore, the tree produced by the synthesis algorithm has

at most $|T|$ leaf nodes. The parallel, choice, sequence and iteration synthesis rules decompose the input net into at least two subnets. Hence, the total number of nodes in the tree[2], is bounded by $2 \cdot |T| - 1$ Therefore, given the input net, Σ, there will be at most $2 \cdot |T| - 1$ applications of the SYNTHESISE procedure, with the time taken for each application bounded by $c \cdot n^4$, for some constant, c. Hence, the time complexity of SYNTHESISE is $O(n^5)$.

The time taken to produce an expression from the tree data structure is $O(n)$. Therefore, the time complexity of BOX EXPRESSION SYNTHESIS, dominated by the call to SYNTHESISE, is $O(n^5)$.

3.5.2 Non-determinism

The analysis of the synthesis algorithm in Section 3.4 highlights three areas of non-determinism: The bracketing order of (sub)expressions whose main connective is $\Box$, ;, or $||$, the ordering of the subexpressions of choice and parallel (sub)expressions, and the choice between the two connected components in the partially decomposed iteration net, Σ_b. In this section, each of these points of non-determinism is discussed in more detail.

The synthesis algorithm produces an expression tree which abstracts away from a bracketing order for the associative operators, $||, \Box$ and ;. The EXPRESSION function imposes a right-associative bracketing order, to produce a properly bracketed expression. For example, an implementation of the expression

$$E = ((a \parallel b) \parallel (c \parallel d)) \parallel e$$

could be synthesised to the equivalent expression:

$$E' = a \parallel (b \parallel (c \parallel (d \parallel e)))$$

[2]Assuming $|T|$ leaf nodes, and a binary decomposition. If an internal node has more than two children, then there will be fewer than $2 \cdot |T| - 1$ nodes in total.

There are 42 possible bracketing orders[3], of E, and all of them give an expression equivalent to E.

In the parallel and choice synthesis rules, an arbitrary ordering of a set of equivalence classes is chosen. This determines the ordering of the subnets in the list field of the node being decomposed, and hence the ordering of the subexpressions in the synthesised expression. For example, an implementation of the expression E, above could be synthesised to the equivalent expression:

$$E'' = d \parallel (b \parallel (e \parallel (c \parallel a)))$$

There are 120 (=5!) different orderings of the subexpressions a, b, c, d and e in E. Each ordering gives an expression equivalent to E.

In the iteration synthesis rule, the partial decomposition, Σ_b is a net consisting of two connected components. One component is chosen at random, and the other is discarded. Proposition 15 demonstrates that the two connected components of Σ_b are isomorphic to each other. Therefore, whichever component is chosen, there will be no difference in the synthesised expression, other than, perhaps, the ordering of subexpressions.

The analysis of the non-determinism in the synthesis algorithm allows the number of expressions equivalent to a given expression to be computed. For example, the size of the equivalence class, to which the expression E belongs is given by $42 \times 120 = 5040$.

3.5.3 Canonical form

A standard form for box expressions is described, and extended to an ordered standard form by introducing a total ordering over expressions. Imposing a fixed bracketing order, such as the right-associative scheme used by EXPRESSION, on an ordered standard form expression, gives a canonical form expression. A modification of the synthesis algorithm, to give a solution to

[3]Given by the 5th Catalan number, $C(n) = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix}$.

CANONICAL BOX EXPRESSION SYNTHESIS is described. Finally, an implementation for CANONICAL BOX EXPRESSION is presented, which rearranges a box expression into its canonical form.

Standard form

The standard form for box expressions abstracts away from a bracketing order for the associative operators, $\|, \square$ and $;$. Hence the standard form of, for example:

$$E = (a \parallel (b \parallel c)) \square (((d; e); (f; g)) \square h)$$

is given by:

$$E' = (a \parallel b \parallel c) \square (d; e; f; g) \square h$$

The SYNTHESISE procedure of the synthesis algorithm finds a standard form expression, which is bracketed by EXPRESSION. Therefore, by Theorem 1, for any box expression, E, the standard form of E, is an unambiguous representation of E. Repeatedly applying the term rewriting rules in Table 3.5 allows any box expression to be rewritten into standard form.

$$\begin{aligned}
E_1 \parallel (E_2 \parallel E_3) &\rightarrow E_1 \parallel E_2 \parallel E_3 \\
(E_1 \parallel E_2) \parallel E_3 &\rightarrow E_1 \parallel E_2 \parallel E_3 \\
E_1 \square (E_2 \square E_3) &\rightarrow E_1 \square E_2 \square E_3 \\
(E_1 \square E_2) \square E_3 &\rightarrow E_1 \square E_2 \square E_3 \\
E_1; (E_2; E_3) &\rightarrow E_1; E_2; E_3 \\
(E_1; E_2); E_3 &\rightarrow E_1; E_2; E_3
\end{aligned}$$

Table 3.5: Rules for rewriting an expression into standard form

Ordered standard form

A standard form expression has choice and parallel subexpressions of the form:

$$E_1 \parallel E_2 \parallel ... \parallel E_k$$

$$E_1 \,\square\, E_2 \,\square\, ... \,\square\, E_k$$

for some $k \geq 2$. The analysis of the synthesis algorithm shows that for any ordering of $E_1, ..., E_k$, the Petri box corresponding to the expression is the same. Imposing a particular ordering on $E_1, ..., E_k$ in such (sub)expressions results in an ordered standard form expression. An ordering of $E_1, ..., E_k$ can be obtained by defining a total order, $<_{\mathsf{e}}$, over expressions, and finding a permutation $E'_1, ..., E'_k$ of $E_1, ..., E_k$ such that for $1 \leq i < k$, $E'_i \leq_{\mathsf{e}} E'_{i+1}$, where for expressions E and F:

$$E \leq_{\mathsf{e}} F \Leftrightarrow E <_{\mathsf{e}} F \vee E = F$$

Firstly, an ordering, $<_{\mathsf{A}}$ over atomic actions is defined. Let $<_{\mathsf{b}}$ be any fixed ordering over the set of basic actions, $\mathcal{B}$. A unique word, $\mathcal{A}(\alpha) \in \mathcal{B}^*$ can be associated with each atomic action, α by writing the basic actions in α in order defined by $<_{\mathsf{b}}$. For any atomic actions, α_1 and α_2:

$$\alpha_1 <_{\mathsf{A}} \alpha_2 \Leftrightarrow \mathcal{A}(\alpha_1) <_{lex} \mathcal{A}(\alpha_2)$$

where $<_{lex}$ is a lexicographic ordering, using $<_{\mathsf{b}}$. For example, suppose $<_{\mathsf{b}}$ is such that $a <_{\mathsf{b}} \widehat{a} <_{\mathsf{b}} b <_{\mathsf{b}} \widehat{b} <_{\mathsf{b}} c <_{\mathsf{b}} ...$, then for atomic actions, $\alpha_1 = \{b, \widehat{a}, c, \widehat{a}\}$, $\alpha_2 = \{\widehat{a}, \widehat{b}\}$, and $\alpha_3 = \{a, c, \widehat{a}\}$:

$$\begin{aligned} \mathcal{A}(\alpha_1) &= \widehat{a}\widehat{a}bc \\ \mathcal{A}(\alpha_2) &= \widehat{a}\widehat{b} \\ \mathcal{A}(\alpha_3) &= a\widehat{a}c \end{aligned}$$

Hence, by the definition of $<_{\mathsf{A}}$: $\alpha_3 <_{\mathsf{A}} \alpha_1 <_{\mathsf{A}} \alpha_2$.

Let $<_{\mathsf{op}}$ be any fixed ordering on the types of box expressions – for example:

$$\text{atomic} <_{\mathsf{op}} \text{parallel} <_{\mathsf{op}} \text{choice} <_{\mathsf{op}} \text{sequence} <_{\mathsf{op}} \text{iteration}$$

$<_e$ is defined inductively, with comparison between atomic actions as the base case:

$$\alpha_1 <_e \alpha_2 \Leftrightarrow \alpha_1 <_A \alpha_2$$

Let F and G be non-atomic standard form expressions, with types op_1 and op_2 and subexpressions $F_1, ..., F_m$ and $G_1, ..., G_n$ respectively. By induction $<_e$ is defined for the subexpressions of F and G. Therefore, without loss of generality, it can be assumed that if the type of F (G) is choice or parallel, then $F_1, ..., F_m$ ($G_1, ..., G_n$) are such that for $1 \leq i < m$, $F_i \leq_e F_{i+1}$ (for $1 \leq i < n$, $G_i \leq_e G_{i+1}$) – *i.e.* if the subexpressions do not have this ordering, then they can be rearranged, using $<_e$ into such an order. Expressions F and G can be compared as follows:

$$\begin{aligned} F <_e G \Leftrightarrow \quad & op_1 <_{op} op_2 \\ \vee \quad & (op_1 = op_2 \wedge m < n) \\ \vee \quad & (op_1 = op_2 \wedge m = n \wedge (\exists i \leq m : (\forall j < i : F_j = G_j) \wedge \\ & F_i <_e G_i)) \end{aligned}$$

Arbitrarily bracketed expressions, E_a and E_b can be compared, by applying $<_e$ to the standard forms E'_a and E'_b of E_a and E_b.

The definition of ordered standard form follows from $<_e$. For a standard form box expression, E, the ordered standard form, $Ord(E)$, is given by:

$$Ord(E) = \begin{cases} \alpha & \text{if } E = \alpha \\ E'_1 \parallel E'_2 \parallel ... \parallel E'_k & \text{if } E = E_1 \parallel E_2 \parallel ... \parallel E_k \\ E'_1 \,\square\, E'_2 \,\square\, ... \,\square\, E'_k & \text{if } E = E_1 \,\square\, E_2 \,\square\, ... \,\square\, E_k \\ Ord(E_1); Ord(E_2); ...; Ord(E_k) & \text{if } E = E_1; E_2; ..; E_k \\ [Ord(E_1) * Ord(E_2) * Ord(E_3)] & \text{if } E = [E_1 * E_2 * E_3] \end{cases}$$

where $E'_1, ..., E'_k$ is a permutation of $Ord(E_1), ..., Ord(E_k)$ such that $E'_i \leq_e E'_{i+1}$, for $1 \leq i < k$.

Canonical form

A canonical expression can be obtained from an ordered standard form expression by imposing a fixed bracketing order on (sub)expressions of the form E_1 *op* E_2 *op* ... *op* E_{k-1} *op* E_k, where $k > 1$, and *op* $\in \{||, \square, ;\}$. A suitable bracketing order is the right-associative scheme:

$$E_1 \; op \; (E_2 \; op \; (... \; op \; (E_{k-1} \; op \; E_k)...))$$

This is the bracketing order that the EXPRESSION function produces.

Canonical box expression synthesis

A small modification to the synthesis algorithm described in Sections 3.2 and 3.3 provides a solution to CANONICAL BOX EXPRESSION SYNTHESIS. The pseudo-code for the modified algorithm is presented below:

```
CANONICAL BOX EXPRESSION SYNTHESIS(Σ)
1   N=new node
2   N.net=Σ
3   ORDERED SYNTHESISE(N)
4   return EXPRESSION(N)

ORDERED SYNTHESISE(N)
1   N.type=ANALYSE(N.net)
2   case N.type
3         atomic: ATOMIC(N)
4         parallel: PARALLEL(N)
5         choice: CHOICE(N)
6         iteration: ITERATION(N)
7         sequence: SEQUENCE(N)
8   for each node N' in N.list
```

```
9        do ORDERED SYNTHESISE(N')
10   if N.type=parallel or choice
11       then sort(N.list)
```

The additional work performed by CANONICAL BOX EXPRESSION SYNTHESIS, to find a canonical form expression does not affect the overall time complexity of the algorithm, which remains at $O(n^5)$.

Canonical box expression

The ideas used in CANONICAL BOX EXPRESSION SYNTHESIS can be applied to finding the canonical form of a box expression, without constructing an implementation of the expression. It is not possible to obtain an efficient solution to CANONICAL BOX EXPRESSION by constructing an implementation of the input expression, and using it as input to CANONICAL BOX EXPRESSION SYNTHESIS, because the size of the implementation of an expression can be exponential in the size of the expression itself. For example, any implementation of:

$$(a \parallel ... \parallel a) \,\Box\, (a \parallel ... \parallel a) \,\Box\, ... \,\Box\, (a \parallel ... \parallel a)$$

has an exponential number of places, and any implementation of an expression with n levels of nested iteration has at least 2^n transitions.

The algorithm for CANONICAL BOX EXPRESSION uses an expression tree corresponding to the standard form of the input expression. A node of this tree is similar to that used in the synthesis algorithm, Figure 3.1, except that the net field is omitted. The standard form is rearranged into ordered standard form by the method used in ORDERED SYNTHESISE. A properly bracketed, canonical form expression is obtained from the ordered standard form, using EXPRESSION.

```
CANONICAL BOX EXPRESSION(E)
1    N=expression tree corresponding to standard form of E
```

```
2   VISIT(N)
3   return EXPRESSION(N)

VISIT(N)
1   if N.type≠atomic
2     for each node N' in N.list
3         do VISIT(N')
4     if N.type=parallel or choice
5       then sort(N.list)
```

Let a be the number of atomic actions in a box expression, E. The time complexity of the VISIT procedure is $O(a^2 \cdot \log a)$, because there are at most a nodes in the expression tree, and for each node, N, the size of $N.list$ is at most a. To sort a list of size a requires $a \cdot \log a$ comparisons, each of which requires $O(1)$ time (assuming that the size of the multiset of basic actions in each atomic action is bounded by some constant).

Uniqueness of canonical form

The analysis of the non-determinism of the synthesis algorithm in Section 3.5.2 led to the definition of a canonical form, and modifications to the synthesis algorithm so that it produces deterministic results. The resulting CANONICAL BOX EXPRESSION SYNTHESIS algorithm is used to show that there is a unique canonical form box expression associated with each Petri box.

Proposition 16 *Let Σ be an implementation of a box expression, E. The expressions produced by* CANONICAL BOX EXPRESSION SYNTHESIS*(Σ) and* CANONICAL BOX EXPRESSION*(E) are in canonical form.*

Proof: By the algorithms for CANONICAL BOX EXPRESSION SYNTHESIS, and CANONICAL BOX EXPRESSION, and the definition of $Ord(E)$, an expression tree corresponding to the ordered standard form of E is produced. The EXPRESSION function, produces an expression, C, with

right-associative bracketing order from this tree. By the definition of the canonical form, C is in canonical form. □

Proposition 17 *Let Σ_x and Σ_y be implementations of box expressions, such that $\Sigma_x =_{iso} \Sigma_y$. The expressions, C_1 and C_2 obtained from calls to* CANONICAL BOX EXPRESSION SYNTHESIS *with nets Σ_x and Σ_y respectively are such that $C_1 = C_2$.*

Proof: By induction on the number of transitions in Σ_x and Σ_y, it is shown that the ordered standard form expressions E_x and E_y, corresponding to the expression trees synthesised from Σ_x and Σ_y respectively, are such that $E_x = E_y$. By the definition of $=_{iso}$, the number of transitions in Σ_x is the same as the number of transitions in Σ_y. The proof relies on the fact that all of the definitions and properties defined in Section 2.5, do not rely on transition or place names – *i.e.* their effect on isomorphic nets is identical.

Base case: Σ_x and Σ_y contain one transition. By Propositions 3 and 8, the atomic action synthesis rule will be applied to Σ_x and Σ_y, to produce expressions α_1 and α_2. By the definition of $=_{iso}$, $\alpha_1 = \alpha_2$.

Induction step: Σ_x and Σ_y contain n transitions, for some $n > 1$. By Proposition 8, and the definitions of Properties 1-4, the same synthesis rule will be applied to both Σ_x and Σ_y:

Parallel: By (3.3), and definitions of $\overset{\leftrightarrow}{\sim}_{N_a}$ and $=_{iso}$, Σ_x and Σ_y are decomposed into nets $\Sigma_1, ..., \Sigma_k$ and $\Sigma'_1, ..., \Sigma'_k$ respectively, such that there exists a permutation, $\theta : 1..k \rightarrow 1..k$, with $\Sigma_i =_{iso} \Sigma'_{\theta(i)}$, for $1 \leq i \leq k$.

Choice: By the definitions of P_{T_e} and $=_{iso}$, Σ_x and Σ_y are decomposed into k subnets, for some $k > 1$. By (3.6), (3.7) and (3.8), and the definition of $=_{iso}$, Σ_x and Σ_y are decomposed into nets $\Sigma_1, ..., \Sigma_k$

and $\Sigma'_1, ..., \Sigma'_k$ respectively, such that there exists a permutation, $\theta : 1..k \rightarrow 1..k$, with $\Sigma_i =_{iso} \Sigma'_{\theta(i)}$, for $1 \leq i \leq k$.

Sequence: By (3.9), and definition of $=_{iso}$, the set of clusters of internal places, S_i, has the same cardinality, k, for both Σ_x and Σ_y. Hence Σ_x and Σ_y are decomposed into the same number of subnets. By the definitions of $<_s$ and $=_{iso}$, the ordered set of clusters $x_1, ..., x_k$ for Σ_x, and $y_1, ..., y_k$ for Σ_y are such that for $1 \leq i \leq k$, $x_i =_{iso} y_i$. Therefore, by (3.12), (3.13), and (3.14), and definition of $=_{iso}$, Σ_x is decomposed into nets $\Sigma_1, ..., \Sigma_k$, and Σ_y is decomposed into nets $\Sigma'_1, ..., \Sigma'_k$ such that for $1 \leq i \leq k$, $\Sigma_i =_{iso} \Sigma'_i$.

Iteration: Let Σ_{bx} and Σ_{by} be the partial decompositions of Σ_x and Σ_y respectively, corresponding to Σ_b in the description of the iteration synthesis rule. Similarly for Σ_{cx} and Σ_{cy}, corresponding to Σ_c. By (3.15), (3.16), (3.17), and definitions of Σ_a, Σ_b and $=_{iso}$: $\Sigma_{bx} =_{iso} \Sigma_{by}$. Proposition 15 shows that the two components of Σ_{bx} and Σ_{by} are isomorphic to each other. Therefore, $\Sigma_{cx} =_{iso} \Sigma_{cy}$. By (3.18), (3.19), (3.20), and (3.21), and the definition of $=_{iso}$, Σ_x and Σ_y are decomposed into nets $\Sigma_1, \Sigma_2, \Sigma_3$, and $\Sigma'_1, \Sigma'_2, \Sigma'_3$ respectively, such that for $1 \leq i \leq k$, $\Sigma_i =_{iso} \Sigma'_i$.

The same synthesis rule is applied to both Σ_x and Σ_y. In the decomposition performed by the synthesis rule applied to Σ_x and Σ_y, no new transitions are created. Therefore, by Proposition 3, the subnets Σ_i and Σ'_i, for $1 \leq i \leq k$ produced by the decomposition of the synthesis rule each contain fewer than n transitions. Hence, by the induction hypothesis, for $1 \leq i \leq k$, the ordered standard form expressions E_i and E'_i, synthesised from Σ_i and Σ'_i are such that:

$$E_i = \begin{cases} E'_{\theta(i)} & \text{if rule applied is parallel or choice} \\ E'_i & \text{otherwise} \end{cases}$$

Suppose the parallel or choice synthesis rule was applied. Let $F_1, ..., F_k$ and $F'_1, ..., F'_k$ be the reordering of the sets of expressions $E_1, ..., E_k$ and $E'_1, ..., E'_k$ according to $<_e$. This reordering corresponds to lines 10 and 11 of ORDERED SYNTHESISE. By the definition of $<_e$, for $1 \leq i \leq k$, $F_i = F_{i'}$. Therefore, the fully refined expressions E_x and E_y, synthesised from Σ_x and Σ_y are such that $E_x = E_y$. The EXPRESSION function, imposes a right-associative bracketing order on E_x and E_y to produce expressions C_1 and C_2. Therefore $C_1 = C_2$. □

Proposition 18 *Let E_1 and E_2 be box expressions, with canonical forms C_1 and C_2 respectively. The definition of canonical form is such that* $\mathsf{box}(E_1) = \mathsf{box}(E_2)$ *if and only if $C_1 = C_2$.*

Proof: Let Σ_1 and Σ_2 be any implementations of E_1 and E_2 respectively. By Proposition 16, C_1=CANONICAL BOX EXPRESSION SYNTHESIS(Σ_1), and C_2=CANONICAL BOX EXPRESSION SYNTHESIS(Σ_2).

Suppose $\mathsf{box}(E_1) = \mathsf{box}(E_2)$, then by definition of box(), $\Sigma_1 =_{iso} \Sigma_2$. Therefore, by Proposition 17, $C_1 = C_2$.

Suppose $C_1 = C_2$. By Theorem 1, and definition of box(): $\mathsf{box}(C_1) = \mathsf{box}(E_1)$, and $\mathsf{box}(C_2) = \mathsf{box}(E_2)$. Therefore $\mathsf{box}(E_1) = \mathsf{box}(E_2)$. □

3.5.4 Decision problems

The two decision problems, PETRI BOX ISOMORPHISM, and BOX EXPRESSION ISOMORPHISM can be solved using CANONICAL BOX EXPRESSION SYNTHESIS and CANONICAL BOX EXPRESSION respectively. In both cases, two canonical form expressions are found, and compared. The pseudo-code for the two problems is presented below:

PETRI BOX ISOMORPHISM(Σ_1, Σ_2)

1 C_1=CANONICAL BOX EXPRESSION SYNTHESIS (Σ_1)

2 C_2=CANONICAL BOX EXPRESSION SYNTHESIS(Σ_2)
3 **if** $C_1 = C_2$
4 **then return** yes
5 **else return** no

BOX EXPRESSION ISOMORPHISM(E_1, E_2)
1 C_1=CANONICAL BOX EXPRESSION(E_1)
2 C_2=CANONICAL BOX EXPRESSION(E_2)
3 **if** $C_1 = C_2$
4 **then return** yes
5 **else return** no

When comparing atomic actions, α_1 and α_2 in canonical form expressions, C_1 and C_2, the words $\mathcal{A}(\alpha_1)$ and $\mathcal{A}(\alpha_2)$ should be compared. The correctness of the algorithms follow from Proposition 18.

3.5.5 Axiom system

Table 3.6 contains the set of axioms that were referred to by the proof of correctness of the synthesis algorithm, in Section 3.4. All of these axioms are introduced in [6], and their soundness is shown there[4]. In this section, it is shown that the axioms can be used to rearrange any box expression into its canonical form, and therefore, the set of axioms in Table 3.6 is complete.

Proposition 19 *The associativity axioms in Table 3.6 allow any box expression, E, to be rearranged to give an equivalent expression E', which has right-associative bracketing order.*

Proof: By structural induction over the box expression syntax. Note that since the axioms in Table 3.6 are sound, the rearranged expression, E' is equivalent to the original expression, E.

[4]Although the soundness proofs are in the context of renaming equivalence, they also hold for the stronger equivalence of isomorphism.

Associativity	$(E_1; E_2); E_3 = E_1; (E_2; E_3)$
	$(E_1 \,\square\, E_2) \,\square\, E_3 = E_1 \,\square\, (E_2 \,\square\, E_3)$
	$(E_1 \parallel E_2) \parallel E_3 = E_1 \parallel (E_2 \parallel E_3)$
Commutativity	$E_1 \,\square\, E_2 = E_2 \,\square\, E_1$
	$E_1 \parallel E_2 = E_2 \parallel E_1$

Table 3.6: Axioms

Base case: The expression $E = \alpha$ has right-associative bracketing order.

Induction step: If $E = [E_1 * E_2 * E_3]$, then by the induction hypothesis, E_1, E_2, and E_3 can be rearranged, using the associativity axioms, to give right-associatively bracketed expressions E_1', E_2' and E_3'. Hence, E can be rewritten to $[E_1' * E_2' * E_3']$, which has right-associative bracketing order.

If $E = E_1 \; op \; E_2$, where $op \in \{\parallel, \square, ;\}$, then by the induction hypothesis, E_1 can be rearranged to give a right-associatively bracketed expression, E_1'. If E_1' has the form $F_1 \; op \; F_2$, *i.e.* has the same main connective as E, then by the associativity axiom, E can be rewritten in the form $F_1 \; op \; (F_2 \; op \; E_2)$. By the induction hypothesis, the subexpression, $F_2 \; op \; E_2$, can be rearranged to give a right-associatively bracketed expression, F_2', which results in the right-associatively bracketed expression $F_1 \; op \; F_2'$ for E. If the main connective of E_1' is not op, then E_2 can be rearranged into E_2', with right-associative bracketing order. Therefore, E can be rewritten to $E_1' \; op \; E_2'$, which has right-associative bracketing order. □

Proposition 20 *The axioms in Table 3.6 allow any box expression, E, to be rearranged into an equivalent expression, C, such that C is in canonical form.*

Proof: By structural induction over the box expression syntax. The rearranged expression, C is equivalent to the original expression, E, because the axioms in Table 3.6 are sound.

Base case: The expression $E = \alpha$ is in canonical form.

Induction step: By Proposition 19, E can be rearranged into an equivalent expression, E', which has right-associative bracketing order. By the induction hypothesis, the subexpression, E_i of E' can be rearranged, using the axioms in Table 3.6 to give the canonical form expression, E'_i.

- $E' = E_1 \; op \; E_2$, where $op \in \{\|, \square\}$: By the induction hypothesis, E', and therefore, E, can be rewritten to $E'_1 \; op \; E'_2$. By the definition of right-associative bracketing order, E'_1 cannot have main connective op. Therefore, there are four possible cases:

  1. E'_2 does not have main connective op, and $E'_2 <_e E'_1$: By the commutativity axiom:

     $$E_1 \; op \; E_2 \; = \; E_2 \; op \; E_1$$

     E can be rewritten to $E'_2 \; op \; E'_1$, which is in canonical form.

  2. E'_2 does not have main connective op, and $E'_2 \geq_e E'_1$: The expression $E'_1 \; op \; E'_2$ is in canonical form.

  3. E'_2 has the form $F_1 \; op \; F_2$, and $F_1 <_e E'_1$: *i.e.* E can be rewritten to $E'_1 \; op \; (F_1 \; op \; F_2)$. By the associativity and commutativity axioms:

     $$\begin{aligned} E'_1 \; op \; (F_1 \; op \; F_2) \; &= \; (E'_1 \; op \; F_1) \; op \; F_2 && \text{Associativity} \\ &\quad\; (F_1 \; op \; E'_1) \; op \; F_2 && \text{Commutativity} \\ &\quad\; F_1 \; op \; (E'_1 \; op \; F_2) && \text{Associativity} \end{aligned}$$

     By the induction hypothesis, $E'_1 \; op \; F_2$ can be rearranged into canonical form, F, resulting in the canonical form expression, $F_1 \; op \; F$, for E.

4. E_2' has the form F_1 *op* F_2, and $F_1 \geq_e E_1'$: The expression E_1' *op* $(F_1$ *op* $F_2)$ is in canonical form.

- $E' = E_1; E_2$: By the induction hypothesis, and the associativity axiom for sequential composition, E', and therefore, E, can be rewritten to canonical form.
- $E' = [E_1 * E_2 * E_3]$: By the induction hypothesis, E', and therefore, E, can be rewritten to $[E_1' * E_2' * E_3']$, which is in canonical form.

□

Theorem 2 *The axiom system in Table 3.6 is complete.*

Proof: By Proposition 20, for any box expressions E_1 and E_2, the axioms in Table 3.6 can be used to rearrange E_1 and E_2 into canonical forms C_1 and C_2 respectively. By Proposition 18:

$$C_1 = C_2 \Leftrightarrow \mathsf{box}(E_1) = \mathsf{box}(E_2)$$

Hence, if E_1 is equivalent to E_2, there exists a sequence of axiom applications which rearranges E_1 into E_2. Therefore, the axiom system in Table 3.6 is complete. □

3.5.6 Generating proofs

In this section, an algorithm for BOX EXPRESSION ISOMORPHISM PROOF is presented which generates a proof that two box expressions are equivalent, using the axioms in Table 3.6. The algorithm uses CANONICAL PROOF which generates a proof that the input expression is equivalent to its canonical form. CANONICAL PROOF is based on the structure of the proofs of Propositions 19, and 20.

In the following pseudo-code, it is assumed that the variables Proof and T' are accessible globally. Proof is a list of parse trees, and is initialised in line 2 of CANONICAL PROOF to be the parse tree corresponding to the input

expression, E. T' is a global pointer which always points to the root of the parse tree being manipulated by the BRACKET, SORT and ORDER procedures. The statement Proof=Proof+T' appends a copy of the parse tree, T' to the list of parse trees, Proof.

```
CANONICAL PROOF(E)
1   T'= parse tree of E
2   Proof=[T']
3   BRACKET(T')
4   SORT(T')
5   return Proof

BRACKET(T)
1   case T.type
2       atomic: do nothing
3       iteration: BRACKET(T.left)
4                  BRACKET(T.middle)
5                  BRACKET(T.right)
6       sequence,choice,parallel:
7                  BRACKET(T.left)
8                  current=T
9                  while current.left.type=T.type
10                     temp=current.left
11                     current.left=temp.left
12                     temp.left=temp.right
13                     temp.right=current.right
14                     current.right=temp
15                     Proof=Proof+T'
16                     current=current.right
17                 BRACKET(current.right)
```

```
SORT(T)
1   case T.type
2         atomic: do nothing
3         iteration: SORT(T.left)
4                    SORT(T.middle)
5                    SORT(T.right)
6         sequence,choice,parallel:
7                    temp=T
8                    while temp.type=T.type
9                          do SORT(temp.left)
10                            temp=temp.right
11                   SORT(temp)
12  if T.type=parallel or choice
13    then ORDER(T)

ORDER(T)
1   do temp=T
2      while temp.right.type=T.type
3            do if temp.right.left <e temp.left
4                  then swap temp.left and temp.right.left
5                       Proof=Proof+T'
6                temp=temp.right
7      if temp.right <e temp.left
8        then swap temp.left and temp.right
9             Proof=Proof+T'
10  while at least one swap is performed
```

Figure 3.12 shows the tree manipulations carried out in lines 10-14 of BRACKET, and lines 4 and 8 of ORDER, together with the corresponding ex-

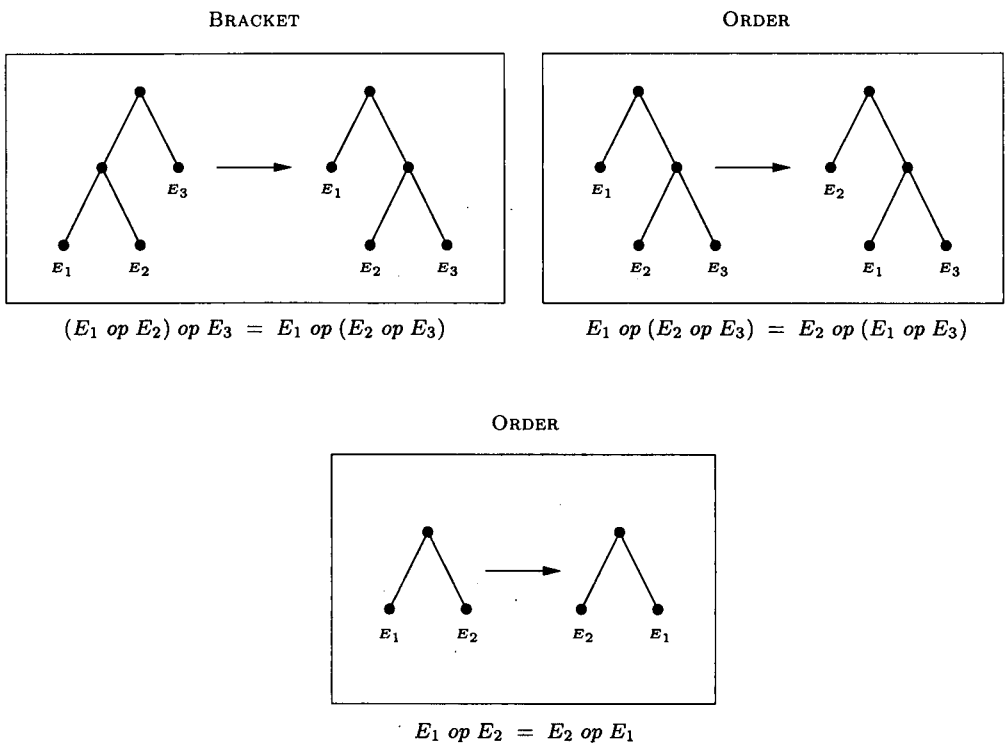


Figure 3.12: Manipulation of the parse tree

pression manipulation. The first and third manipulations correspond directly to an application of the associativity and commutativity axioms respectively. The second manipulation involves both the associativity and commutativity axioms, as follows:

$$\begin{aligned} E_1\ op\ (E_2\ op\ E_3) \;=\; & (E_1\ op\ E_2)\ op\ E_3 && \text{Associativity} \\ & (E_2\ op\ E_1)\ op\ E_3 && \text{Commutativity} \\ & E_2\ op\ (E_1\ op\ E_3) && \text{Associativity} \end{aligned}$$

The correctness of CANONICAL PROOF follows from the correspondence of the code with the structure of the proofs in Propositions 19, and 20. Note that the BRACKET procedure contains an optimisation, because a direct translation of the the proof structure of Proposition 19 results in an inefficient algorithm. The optimisation rearranges an expression of the form $E_1\ op\ E_2$, where $E_1 = (E'_1\ op\ (E'_2\ op\ (...\ op\ (E_{k-1}\ op\ E_k)...)))$, for some $k > 1$ has right-associative

bracketing order into:

$$E'_1 \; op \; (E'_2 \; op \; (... \; op \; (E_{k-1} \; op \; (E_k \; op \; E_2))...))$$

by repeatedly applying the associativity axiom. A right-associatively bracketed expression for, E, can be obtained by rearranging E_2 to have right-associative bracketing order.

Let a be the number of atomic actions in a box expression, E. The time complexity of the BRACKET procedure is $O(a^2)$, and $O(a^2)$ manipulations (axiom applications) are performed. The time complexity of SORT is $O(a^3)$ because there are at most a nodes in the parse tree, and the ORDER procedure has time complexity $O(a^2)$. ORDER is an implementation of bubble sort, and for each call, it sorts at most a subexpressions. Therefore, at most a^2 comparisons are made, each having time complexity $O(1)$ (assuming that the size of the multiset of basic actions in each atomic action is bounded by some constant). Hence the number of axiom applications performed by SORT is $O(a^3)$.

Let E_1, E_2 be box expressions, such that $\mathsf{box}(E_1) = \mathsf{box}(E_2)$. By Proposition 18, the canonical forms of E_1 and E_2 will be the same. Therefore, BOX EXPRESSION ISOMORPHISM PROOF can produce a proof that $E_1 = E_2$ by concatenating the proofs generated by CANONICAL PROOF(E_1) and CANONICAL PROOF(E_2).

BOX EXPRESSION ISOMORPHISM PROOF(E_1, E_2)

1 Proof$_1$=CANONICAL PROOF(E_1)

2 Proof$_2$=CANONICAL PROOF(E_2)

3 Output Proof$_1$

4 Output Proof$_2$ in reverse order.

The time complexity of BOX EXPRESSION ISOMORPHISM PROOF, on input E_1 and E_2 is $O(a^3)$, where $a = \max\{a_1, a_2\}$, and a_1 and a_2 are the number of

atomic actions in E_1 and E_2 respectively. The length of the proof generated by BOX EXPRESSION ISOMORPHISM PROOF is $O(a^3)$.

3.5.7 Examples

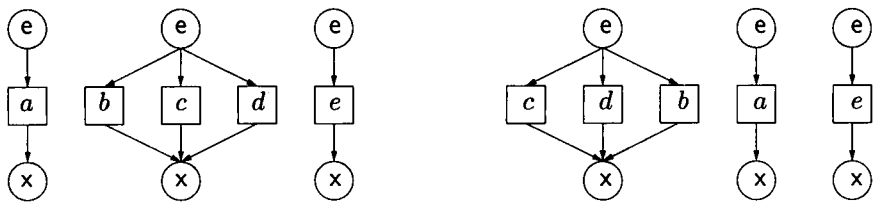


Figure 3.13: Example nets

Figure 3.13 shows two nets, Σ_1 and Σ_2, which are implementations of unknown expressions. The synthesis algorithm can be used to find expressions E_1 and E_2 such that $\mathsf{box}(E_1) = [\Sigma_1]$, and $\mathsf{box}(E_2) = [\Sigma_2]$. The expressions produced by BOX EXPRESSION SYNTHESIS, with inputs Σ_1 and Σ_2, will not necessarily be in canonical form. For example:

$$\begin{aligned} \text{BOX EXPRESSION SYNTHESIS}(\Sigma_1) &= a \parallel ((b \,\Box\, (c \,\Box\, d)) \parallel e) \\ \text{BOX EXPRESSION SYNTHESIS}(\Sigma_2) &= (c \,\Box\, (d \,\Box\, b)) \parallel (a \parallel e) \end{aligned}$$

Using CANONICAL BOX EXPRESSION SYNTHESIS, canonical expressions C_1 and C_2, corresponding to Σ_1 and Σ_2, can be found. The same result is obtained by applying CANONICAL BOX EXPRESSION to the synthesised expressions, E_1 and E_2:

$$\begin{aligned} C_1 &= a \parallel (e \parallel (b \,\Box\, (c \,\Box\, d))) \\ C_2 &= a \parallel (e \parallel (b \,\Box\, (c \,\Box\, d))) \end{aligned}$$

Hence the calls to PETRI BOX ISOMORPHISM(Σ_1, Σ_2) and BOX EXPRESSION ISOMORPHISM(E_1, E_2) both give the output "yes".

BOX EXPRESSION ISOMORPHISM PROOF performs the rearrangement of an expression into canonical form, using the application of a set of axioms. Hence a proof that an expression is equivalent to its canonical form is produced. Given equivalent expressions, E_1 and E_2 as input, the algorithm generates proofs that $E_1 = C_1$ and $E_2 = C_2$, where C_1 and C_2 are the canonical forms of E_1 and E_2 respectively. As E_1 and E_2 are equivalent, $C_1 = C_2$, and a proof that $E_1 = E_2$ can be obtained by concatenating the proof of $E_1 = C_1$ with the proof of $E_2 = C_2$ reversed:

$$
\begin{aligned}
E_1 &= a \parallel ((b \,\square\, (c \,\square\, d)) \parallel e) \\
&= a \parallel (e \parallel (b \,\square\, (c \,\square\, d))) = C_1 = C_2 \\
&= a \parallel ((b \,\square\, (c \,\square\, d)) \parallel e) \\
&= (a \parallel (b \,\square\, (c \,\square\, d))) \parallel e \\
&= ((b \,\square\, (c \,\square\, d)) \parallel a) \parallel e \\
&= (b \,\square\, (c \,\square\, d)) \parallel (a \parallel e) \\
&= ((b \,\square\, c) \,\square\, d) \parallel (a \parallel e) \\
&= ((c \,\square\, b) \,\square\, d) \parallel (a \parallel e) \\
&= (c \,\square\, (b \,\square\, d)) \parallel (a \parallel e) \\
&= (c \,\square\, (d \,\square\, b)) \parallel (a \parallel e) \\
&= E_2
\end{aligned}
$$

In general, the proof will not be the shortest possible. However, it has length at most polynomial in the size of the input expressions.

Chapter 4

Synchronisation synthesis

4.1 Introduction

This chapter considers the extension of the synthesis algorithm of Chapter 3 by including support for the synchronisation operator. The semantics for synchronisation operate globally on the net, which makes the synthesis problem more difficult than for the operators in the basic syntax.

$E ::=$	α	Atomic action
	$\mid E \parallel E$	Parallel composition
	$\mid E \,[]\, E$	Choice composition
	$\mid E; E$	Sequential composition
	$\mid [E * E * E]$	Iteration
	$\mid E$ **sy** A	Synchronisation

Table 4.1: Language defining class of synthesisable Petri boxes

The synthesis problem is to provide an algorithmic translation from an implementation, Σ, of an unknown box expression, to an expression, E, such that any implementation of E is isomorphic to Σ. The aim is to extend the class of Petri boxes allowed as input to the synthesis algorithm presented in Chapter 3, to cope with input nets which are implementations of box expres-

sions involving the synchronisation operator – *i.e.* an implementation of any expression from the syntax in Table 4.1 is allowed as input to the synthesis algorithm presented here. Unless stated otherwise, every box expression should be assumed to be a member of the language generated by the syntax in Table 4.1, and every net an implementation of such a box expression.

BOX EXPRESSION SYNTHESIS

INSTANCE: Net, Σ, member of the class of Petri boxes allowed as input.

SOLUTION: Box expression, E from the syntax in Table 4.1,

such that $\mathsf{box}(E) = [\Sigma]$.

Section 4.2 investigates some of the issues affecting the synthesis of synchronisation, and shows that the synthesis problem is NP-hard. The hardness result arises because of the difficulty of identifying a grouping for the transitions to be represented by a particular synchronisation operator in the expression. Alternative approaches to the synthesis problem in the light of this result are discussed in Section 4.2. Sections 4.3 and 4.4 present a solution to the synthesis problem for the class of input nets that can be obtained from an expression over the syntax in Table 4.1. The solution reuses the synthesis algorithm of Chapter 3. In Section 4.5, the correctness of the synthesis algorithm is shown. The analysis carried out in Section 3.4 forms the basis for the discussion, and solutions to related problems presented in Section 4.6. The complexity of the problem of finding a canonical form is investigated in Section 4.6.3. The investigation in Section 4.6.2 provides a basis for the derivation of a complete axiom system for a fragment of the Petri Box Calculus containing the synchronisation operator, presented in Section 4.6.4.

The semantics for the synchronisation and restriction operators, given in Chapter 1 were given in terms of the synchronisation and restriction by a single basic action. It is notationally convenient (and possible) to be able to synchronise and restrict by a set of basic actions. The ability to use this extended notation follows from the properties:

$$E \ \mathbf{sy}\ a\ \mathbf{sy}\ b \ = \ E \ \mathbf{sy}\ b\ \mathbf{sy}\ a \tag{4.1}$$

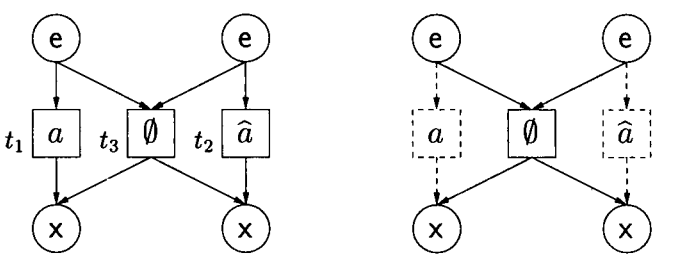
$$E \textbf{ rs } a \textbf{ rs } b \;=\; E \textbf{ rs } b \textbf{ rs } a \tag{4.2}$$

Hence, for any set of basic actions $A = \{a_1, ..., a_n\}$, and expression $E = E'$ **sy** A (respectively $E = E'$ **rs** A), the Petri box, $\mathsf{box}(E)$, can be constructed by applying the semantics below to the equivalent expressions E' **sy** a_1... **sy** a_n (respectively E' **rs** a_1... **rs** a_n). The correctness of (4.2) follows directly from the definition of restriction. The correctness of (4.1) is less obvious, and is discussed further in Section 4.2.2.

$\mathcal{L}(E)$ is defined to be the set of basic actions that appears in the transition labels of an implementation of E. The definition of $\mathcal{L}$ is limited to expressions which do not contain the restriction or scoping operators – *i.e.* for any expression, E, from the syntax in Table 4.1:

$$\mathcal{L}(E) = \begin{cases} \{l \mid l \in \alpha\} & \text{if } E = \alpha \\ \mathcal{L}(E_1) \cup \mathcal{L}(E_2) & \text{if } E = E_1 op E_2 \text{ for } op \in \{\|, \square, ;\} \\ \mathcal{L}(E_1) \cup \mathcal{L}(E_2) \cup \mathcal{L}(E_3) & \text{if } E = [E_1 * E_2 * E_3] \\ \mathcal{L}(E_1) & \text{if } E = E_1 \textbf{ sy } A \end{cases}$$

Figure 4.1 shows implementations of two simple box expressions, involving the synchronisation and restriction operators. Note that by the definition of scoping, the implementation of $(a \parallel \widehat{a})$ **sy** a **rs** a is also an implementation of the expression $[a : a \parallel \widehat{a}]$.



$E = (a \parallel \widehat{a})$ **sy** a — Synchronisation

$E = (a \parallel \widehat{a})$ **sy** a **rs** a — Restriction

Figure 4.1: Synchronisation and restriction

The semantics for synchronisation used here are slightly different from

those of [6], in that originally a candidate synchronisation, τ could consist of a single element. In effect this means that a synchronisation operation performed on a net would create a duplicate of each transition that takes part in a synchronisation. Where duplication equivalence is used as the net semantic, the duplicates that are produced by a synchronisation operation are insignificant. However, the duplicates are significant for isomorphism, and it seems counter-intuitive for the synchronisation operation to produce them. It would require only minor modifications to the synthesis algorithm presented in Sections 4.3 and 4.4 to cope with the original semantics of synchronisation given in [6], should it be necessary to do so.

4.2 Synchronisation

In this section an alternative semantics for the synchronisation operator are presented, some general properties of synchronisation are discussed, and the factors affecting the synthesis of expressions from nets containing synchronisation are investigated. In Section 4.2.4, BOX EXPRESSION SYNTHESIS is shown to be NP-hard. Section 4.2.5 discusses various approaches to dealing with the hardness result and concludes by restating BOX EXPRESSION SYNTHESIS in a form that allows an efficient solution.

4.2.1 Semantics of synchronisation

The semantics for synchronisation presented in Section 1.3 is based on a decision procedure which determines whether a particular multiset of transitions constitutes a synchronisation or not. In this section, an iterative version of the semantics for the synchronisation operation is described. The purpose of introducing an alternative semantics is to provide greater intuition into the workings of the synchronisation operator, and to allow the observation that every synchronised transition can be viewed as arising from the synchronisation of a pair of transitions.

A multiset of transitions, $\gamma(t)$ is associated with each transition t. Initially:

$$\forall t \in T : \gamma(t) = \{t\}$$

The purpose of γ is to ensure the correct number of synchronised transitions are created. If the duplication equivalence of [6] is used in place of isomorphism, then γ is not required, because duplicates are not significant. When synchronising an implementation of a box expression on a basic action, a:

1. Find a pair of transitions, t_1 and t_2 with $a \in \lambda(t_1)$ and $\widehat{a} \in \lambda(t_2)$, such that there is no transition, t with $\gamma(t) = \gamma(t_1) + \gamma(t_2)$.

2. Synchronise t_1 and t_2 to obtain a new transition t with:

$$\begin{aligned} W(t,s) &= W(t_1,s) + W(t_2,s) \quad (\text{for } s \in S \cup T) \\ W(s,t) &= W(s,t_1) + W(s,t_2) \quad (\text{for } s \in S \cup T) \\ \lambda(t) &= (\lambda(t_1) + \lambda(t_2)) - \{a, \widehat{a}\} \\ \gamma(t) &= \gamma(t_1) + \gamma(t_2) \end{aligned}$$

3. Repeat the process until no new synchronisations are available.

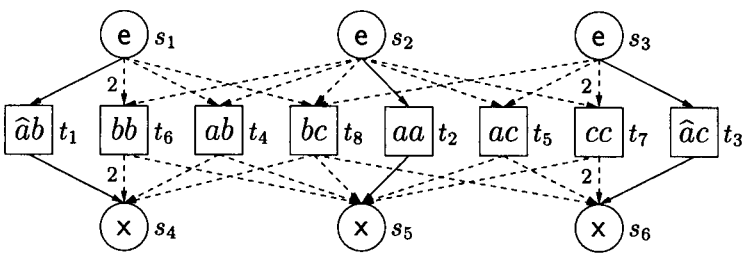


Figure 4.2: Example of the iterative synchronisation scheme

The new transitions created in Step 2 are themselves candidates for synchronisation, provided they have an a or $\widehat{a}$ action in their labels. It is possible for a transition to synchronise with itself (*i.e.* $t_1 = t_2$ in Step 1). If such a synchronisation occurs, then the synchronisation process will be infinite. Infinite synchronisations are discussed in Section 4.2.2. Figure 4.2 shows an

implementation of the expression:

$$E = (\{a, a\} \parallel \{\widehat{a}, b\} \parallel \{\widehat{a}, c\})\ \mathbf{sy}\ a$$

The transitions arising from the synchronisation operator are shown with dotted arcs and boxes. The following table illustrates the synchronisation process:

synchronisation	new transition	λ	γ
t_1, t_2	t_4	$\{a, b\}$	$\{t_1, t_2\}$
t_2, t_3	t_5	$\{a, c\}$	$\{t_2, t_3\}$
t_1, t_4	t_6	$\{b, b\}$	$\{t_1, t_1, t_2\}$
t_3, t_5	t_7	$\{c, c\}$	$\{t_2, t_3, t_3\}$
t_1, t_5	t_8	$\{b, c\}$	$\{t_1, t_2, t_3\}$

In the final row of the table, the transitions t_3 and t_4 could have been synchronised instead of t_1 and t_5, to obtain identical results. Once t_1 and t_5 have been synchronised, the synchronisation of t_3 and t_4 is prevented because $\gamma(t_3) + \gamma(t_4) = \{t_1, t_2, t_3\}$, which is the same as $\gamma(t_8)$. Using the iterative semantics for synchronisation the transitions t_4 and t_5 in Figure 4.2 must be created before the transitions t_6, t_7 and t_8, while the semantics of Section 1.3 allows every synchronised transition to be created independently.

The following proposition shows that the two alternative semantics for synchronisation are consistent with each other.

Proposition 21 *For every box expression, E from the syntax in Table 4.1, and basic action, a, let Σ_1 and Σ_2 be implementations of E* **sy** *a constructed, respectively, using the semantics for synchronisation of Section 1.3 , and using the iterative semantics for synchronisation presented here - then $\Sigma_1 =_{iso} \Sigma_2$.*

Proof: Follows from Lemma 6.3 in [6]. □

4.2.2 Properties of synchronisation

[6] shows that, in the context of duplication equivalence, the synchronisation operator is both idempotent and commutative. The idea of the proof for com-

mutativity of **sy** in [6] holds for the slightly different semantics for synchronisation used here, and in context of isomorphism. For example, any implementations of $E_1 = (\{a, \widehat{c}\} \parallel (\widehat{a}; c))$ **sy** a **sy** c and $E_2 = (\{a, \widehat{c}\} \parallel (\widehat{a}; c))$ **sy** c **sy** a are isomorphic. Figure 4.3 shows a net which is an implementation of both E_1 and E_2. Applying the iterative semantics of Section 4.2.1 illustrates how the transition t_5 in Figure 4.3 arises in different ways for E_1 and E_2:

expression	operation	synchronisation	new transition	λ	γ
E_1	**sy** a	t_1, t_2	t_4	$\{\widehat{c}\}$	$\{t_1, t_2\}$
E_1	**sy** c	t_1, t_3	t_6	$\{a\}$	$\{t_1, t_3\}$
E_1	**sy** c	t_3, t_4	t_5	$\emptyset$	$\{t_1, t_2, t_3\}$
E_2	**sy** c	t_1, t_3	t_6	$\{a\}$	$\{t_1, t_3\}$
E_2	**sy** a	t_1, t_2	t_4	$\{\widehat{c}\}$	$\{t_1, t_2\}$
E_2	**sy** a	t_2, t_6	t_5	$\emptyset$	$\{t_1, t_2, t_3\}$

The commutativity of the synchronisation operator means that it is possible to synchronise on a set of basic actions without ambiguity. Theorem 6.4 (i) in [6] shows that commutativity holds in the context of duplication equivalence. The idea of the proof also works when the net semantic used is isomorphism.

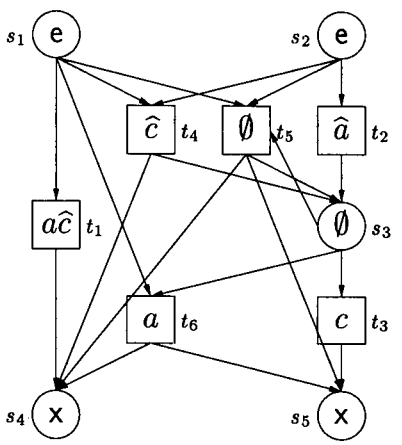


Figure 4.3: Commutativity of synchronisation

The synchronisation operator is not idempotent in the context of isomorphism. This is because synchronising twice produces a second set of syn-

chronised transitions which are significant for isomorphism. For example, Figure 4.4 shows Σ_1, and Σ_2, implementations of $(a \parallel \widehat{a})$ **sy** a and $(a \parallel \widehat{a})$ **sy** a **sy** a respectively. While Σ_1 and Σ_2 are duplication equivalent, they are not isomorphic.

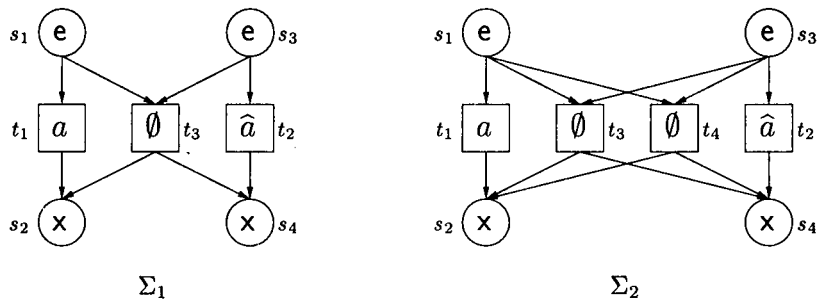


Figure 4.4: Idempotence of synchronisation

Multi-way synchronisation

A multi-way synchronisation is obtained when a transition arising from a synchronisation operation is involved in a further synchronisation. An indirect multi-way synchronisation occurs when the synchronised transition, and further synchronisation are obtained from different applications of the **sy** operator. The multi-way synchronisation is direct when both transitions arise from the same application of a **sy** operator. For example, in Figure 4.2, the transitions t_6, t_7 and t_8 are direct 3-way synchronisations, while in Figure 4.3, transition t_5 is an indirect 3-way synchronisation.

When synchronising on a basic action, a, a direct multi-way synchronisation can only be obtained if there exists a transition, t, in the scope of the **sy** operator, with $\lambda(t)(a) > 1$ or $\lambda(t)(\widehat{a}) > 1$. For example, t_2 in Figure 4.2 has a label containing two a actions. Hence, the transitions t_4 and t_5, arising respectively from the synchronisations t_1, t_2 and t_2, t_3, both have labels containing an a action, and can therefore take part in further synchronisations.

To produce an indirect n-way synchronisation, there must be a transition, t which is in the scope of n **sy** operations, on different basic actions, $b_1, ..., b_n$

for some $n > 1$, such that for $1 \leq i \leq n$, $b_i \in \lambda(t)$, or $\widehat{b_i} \in \lambda(t)$. For example, in Figure 4.3, the transition, t_1 is the the scope of synchronisations on the basic actions, a and c, $a \in \lambda(t_1)$, and $\widehat{c} \in \lambda(t_1)$.

Two restrictions on the form of atomic actions, which affect the type of synchronisations that can be obtained are described below:

- If the action is restricted from being a multiset of basic actions to being a set of basic actions, with the corresponding modifications to the semantics of synchronisation, then it is not possible to obtain a direct multi-way synchronisation. However, indirect multi-way synchronisations can still be produced.

- Further restricting actions to be either a single basic action, or $\emptyset$ prevents the creation of either type of multi-way synchronisation. With this restriction, every transition arising from a synchronisation operation has the label, $\emptyset$.

Intuitively, imposing one of these restrictions could make the synthesis problem easier. However, it will be shown in Section 4.2.4 that the computational complexity of the problem is not affected by either of the restrictions.

Infinite synchronisation

Under certain circumstances, an application of the **sy** operator can produce infinitely many synchronised transitions. Using the iterative semantics for synchronisation, presented in Section 4.2.1, an infinite synchronisation is detected when a synchronisation is performed which produces a new transition with the same label as one of the synchronising transitions. For example if t_1 and t_2 synchronise to give a new transition, t, such that $\lambda(t) = \lambda(t_1)$, then it will be possible to synchronise t and t_2, and so on. The simplest example of an infinite synchronisation occurs when a transition can synchronise with itself: $E_1 = \{a, \widehat{a}\}$ **sy** a. A more complex example is E_2 **sy** $\{a, b\}$, where $E_2 = (\{a, a\} \parallel \{\widehat{a}, b\} \parallel \{\widehat{a}, \widehat{b}\})$. Figure 4.5 shows implementations of both E_1

and E_2. Table 4.2, demonstrates that an infinite synchronisation is detected, when applying the iterative semantics for synchronisation to E_2 **sy** $\{a, b\}$.

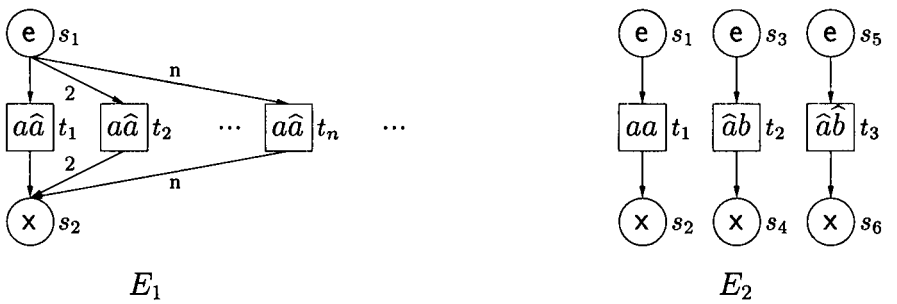


Figure 4.5: Infinite synchronisations

operation	synchronisation	new transition	λ	γ	∞- **sy**
sy a	t_1, t_2	t_4	$\{a, b\}$	$\{t_1, t_2\}$	no
sy a	t_1, t_3	t_5	$\{a, \widehat{b}\}$	$\{t_1, t_3\}$	no
sy a	t_2, t_4	t_6	$\{b, b\}$	$\{t_1, t_2, t_2\}$	no
sy a	t_3, t_4	t_7	$\{b, \widehat{b}\}$	$\{t_1, t_2, t_3\}$	no
sy a	t_3, t_5	t_8	$\{\widehat{b}, \widehat{b}\}$	$\{t_1, t_3, t_3\}$	no
sy b	t_2, t_3	t_9	$\{\widehat{a}, \widehat{a}\}$	$\{t_2, t_3\}$	no
sy b	t_2, t_5	t_{10}	$\{a, \widehat{a}\}$	$\{t_2, t_5\}$	no
sy b	t_2, t_7	t_{11}	$\{\widehat{a}, b\}$	$\{t_2, t_7\}$	yes

Table 4.2: Detecting infinite synchronisations

It is impractical to consider infinite nets as input to the synthesis algorithm. Hence, it is not possible to synthesise expressions from the entire class of implementations of box expressions from the syntax in Table 4.1. This implies that, unlike the synthesis algorithm described in Chapter 3, an analysis of the synthesis algorithm for synchronisation cannot provide a complete axiomatisation of the subset of the Petri Box Calculus defined by the syntax in Table 4.1.

An important point regarding infinite synchronisations is that from a behavioural view, only a finite number of the infinite number of synchronised transitions are significant, and the remainder can never be enabled. Therefore, it may be possible to modify the synchronisation operator semantics so that insignificant transitions, from a behavioural point of view, are not created. However, for structural semantics, such as isomorphism, every transition created by an infinite synchronisation is significant.

4.2.3 Synthesis with synchronisation

In this section, some properties of the synchronisation operator are investigated. This investigation is intended to give some insight into the synthesis problem, and motivate approaches to a synthesis algorithm for the syntax in Table 4.1, possibly by extending the algorithm described in Chapter 3. The following areas are investigated:

- The overlap between expressions from the basic syntax, and those involving the synchronisation operator.
- Equivalence when synchronising on different basic actions – *i.e.* expressions E such that any implementation of E **sy** a is isomorphic to an implementation of E **sy** b.
- Positioning of the synchronisation operators.

Only synchronisations that create at least one transition in the implementation of the expression are considered. For example, the synchronisation operation in $(a \parallel b)$ **sy** a has no effect, and is therefore not considered.

The underlying expression of an expression involving the synchronisation operator is obtained by removing all instances of **sy** . For example, E_2 is the underlying expression of E_1, where:

$$E_1 = ((a \parallel \widehat{a} \parallel b) \textbf{ sy } a \,\Box\, (c \parallel \widehat{c} \parallel \widehat{b}) \textbf{ sy } c) \textbf{ sy } b$$

$$E_2 = (a \parallel \widehat{a} \parallel b) \,\Box\, (c \parallel \widehat{c} \parallel \widehat{b})$$

One approach to dealing with synchronisation is to try and identify in the input net those transitions which have arisen from synchronisation, and remove them. This would leave an implementation of the underlying expression, which can be synthesised using the basic syntax synthesis algorithm.

Overlap with the basic syntax

The simplest example of the overlap between synchronisation and the basic syntax is given by the expressions $(a \parallel \widehat{a}) \,\square\, \emptyset$ and $(a \parallel \widehat{a})$ **sy** a which have isomorphic implementations. Let F_1 and F_2 be:

$$\begin{aligned} F_1 &= ((a \,\square\, E_1) \parallel (\widehat{a} \,\square\, E_2)) \,\square\, \emptyset \\ F_2 &= ((a \,\square\, E_1) \parallel (\widehat{a} \,\square\, E_2)) \textbf{ sy } a \end{aligned} \tag{4.3}$$

In general, the expressions F_1 and F_2 are equivalent, provided $(\mathcal{L}(E_1) \cup \mathcal{L}(E_2)) \cap \{a, \widehat{a}\} = \emptyset$ – *i.e.* the subexpressions E_1 and E_2 do not contain any a or $\widehat{a}$ basic actions:

The condition that E_1 and E_2 must not contain any a or $\widehat{a}$ actions causes problems in the identification of transitions that have arisen from synchronisation. For example, Figure 4.6 shows an implementation of the expression $E = ((\widehat{a} \,\square\, a) \parallel \widehat{a}) \,\square\, \emptyset$. The transition t_4, at first, appears to have arisen from the synchronisation of t_2 and t_3. However, any synchronisation operation with t_2 and t_3 in scope must also have t_1 and t_2 in scope. The absence of an $\emptyset$ transition t, such that $t \bowtie \{t_1, t_2\}$ indicates that t_4 cannot have arisen from a synchronisation operator. This example demonstrates that a general procedure for identifying those transitions arising from synchronisation can only be achieved by examining the net globally.

The class of equivalent expressions determined by F_1 and F_2 in (4.3) can be extended to multi-actions. For example, Figure 4.7 shows a net which is isomorphic to implementations of G_1 and G_2:

$$\begin{aligned} G_1 &= ((\{a, c\} \,\square\, \{\widehat{b}, c, d\}) \parallel (\{\widehat{a}, d\} \,\square\, b)) \textbf{ sy } a \\ G_2 &= ((((\{a, c\} \,\square\, \{\widehat{b}, c, d\}) \parallel (\{\widehat{a}, d\} \,\square\, b))) \,\square\, \{c, d\} \end{aligned}$$

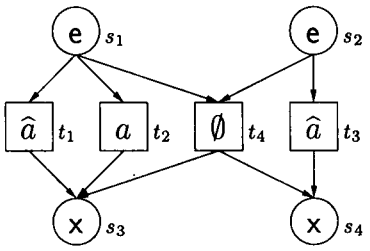


Figure 4.6: Not a synchronisation

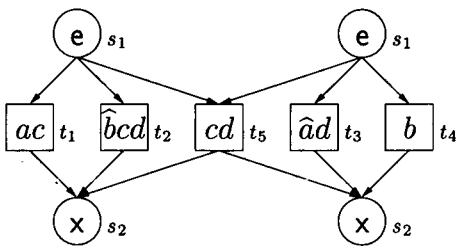


Figure 4.7: Overlap between synchronisation and basic syntax

From the point of view of synthesis, implementations of expressions involving only these limited forms of synchronisation can be synthesised using the basic syntax algorithm described in Chapter 3.

Equivalent synchronisations

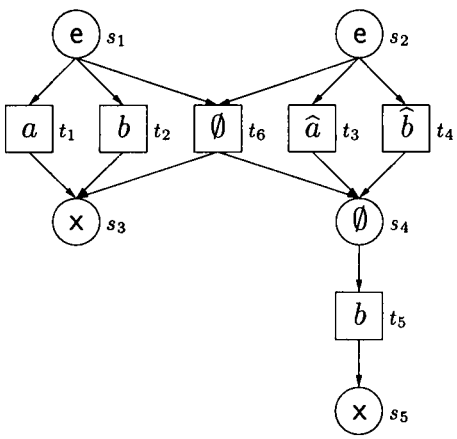
A transition arising from a synchronisation operation inherits the connectivity of the synchronising transitions. Hence, if there are transitions present in the net which have the same connectivity, it may be possible to synchronise on different basic actions to produce equivalent results. For example, the net in Figure 4.7 is isomorphic to implementations of H_1 and H_2:

$$H_1 = (\{a, c\} \,\square\, \{\widehat{b}, c, d\}) \parallel (\{\widehat{a}, d\} \,\square\, b)) \textbf{ sy } a$$

$$H_2 = (\{a, c\} \,\square\, \{\widehat{b}, c, d\}) \parallel (\{\widehat{a}, d\} \,\square\, b)) \textbf{ sy } b$$

In order to represent synchronised transitions which can be obtained from a number of equivalent synchronisations, a synthesis algorithm must choose exactly one of the equivalent synchronisations. As with the overlap between

synchronisation and the basic syntax, identifying equivalent synchronisations entails examining the input net globally. For example, in Figure 4.8, it appears that t_6 can be obtained by a synchronisation operation on either a or b. However, the absence of a synchronisation between t_4 and t_5 means that only the synchronisation on a is valid.



$$((a \,\square\, b) \parallel ((\widehat{a} \,\square\, \widehat{b}); b))\ \mathbf{sy}\ a$$

Figure 4.8: Finding equivalent synchronisations

Position of the sy operators

A synchronisation operation over a basic action, a, has no effect on an expression or subexpression which does not contain any a or $\widehat{a}$ basic actions. Therefore, there is some flexibility in the positioning of synchronisation operators, which preserve equivalence of the corresponding Petri boxes. For example:

$$E_1\ \mathbf{sy}\ a \parallel E_2 = (E_1 \parallel E_2)\ \mathbf{sy}\ a$$

provided that there are no atomic actions containing a or $\widehat{a}$ in E_2 (*i.e.* $\mathcal{L}(E_2) \cap \{a, \widehat{a}\} = \emptyset$). Similarly for the sequence and choice operators. This property does not apply to the iteration operator because the semantics of $[E_1 * E_2 * E_3]$ use two copies of the implementations of E_1, E_2 and E_3. Hence, for example,

$$[(a; \widehat{a})\ \mathbf{sy}\ a * b * c] \neq [a; \widehat{a} * b * c]\ \mathbf{sy}\ a$$

Nets (i) and (ii) in Figure 4.9 are implementations of $[(a;\widehat{a})$ **sy** $a*b*c]$ and $[a;\widehat{a}*b*c]$ **sy** a respectively. When the synchronisation operator is at the top level, extra synchronisations take place between the two copies of $a;\widehat{a}$.

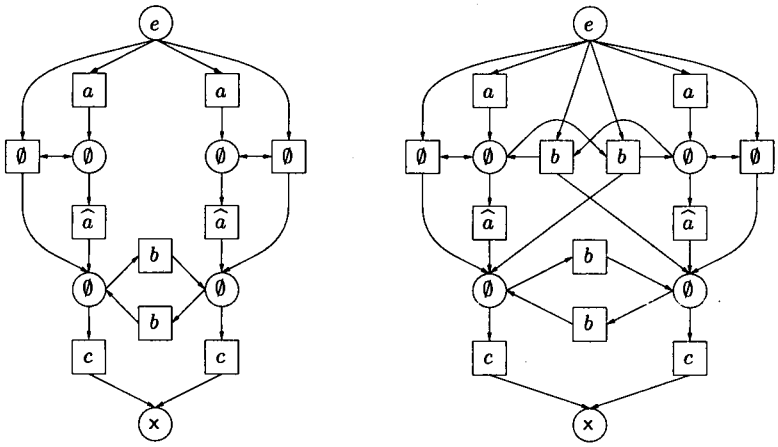


Figure 4.9: Synchronisation and iteration

4.2.4 Synthesis with synchronisation is NP hard

In this section, the complexity of synthesising expressions from nets which are implementations of expressions from the syntax in Table 4.1 is investigated. When the syntax of Table 4.1 is used to represent the synthesised expression, the synthesis problem is shown to be NP hard. Some approaches to deal with this result are discussed in Section 4.2.5. The NP hardness result for the synthesis problem is demonstrated using a transformation from ONE-IN-THREE 3SAT [47, 25] to SYNCHRONISATION ASSIGNMENT.

ONE-IN-THREE 3SAT

INSTANCE: Set, U of variables, collection C of clauses over U such that each clause $c \in C$ has $|c| = 3$.

QUESTION: Is there a truth assignment for U such that each clause in C has exactly one true literal?

SYNCHRONISATION ASSIGNMENT

INSTANCE: Net, Σ, the implementation of a basic syntax box expression, and set, X, of new transitions.

QUESTION: Is the net $\Sigma \oplus (X, \emptyset)$ an implementation of a box expression from the syntax in Table 4.1?

Let $U = \{u_1, ..., u_n\}$, and $C = \{C_1, ..., C_m\}$, where each $C_i = \{c_{i_1}, c_{i_2}, c_{i_3}\}$, be an arbitrary instance of ONE-IN-THREE 3SAT. A corresponding instance of SYNCHRONISATION ASSIGNMENT can be constructed from U and C. Let Σ be an implementation of $E = E_1; E_2; E_3$, where:

$$
\begin{aligned}
E_1 &= ((\widehat{u_1} \,\square\, \widehat{\neg u_1}) \,\square\, ... \,\square\, (\widehat{u_n} \,\square\, \widehat{\neg u_n})...) \\
E_2 &= (((c_{1_1} \,\square\, c_{1_2}) \,\square\, c_{1_3}); ...; ((c_{m_1} \,\square\, c_{m_2}) \,\square\, c_{m_3})...) \\
E_3 &= ((u_1 \,\square\, \neg u_1); ...; (u_n \,\square\, \neg u_n)...)
\end{aligned}
$$

There is a correspondence between the atomic actions in E, and the transitions in Σ (formalised in Section section-action-transitions). Hence $t(u)$, can be used to unambiguously refer to the transition in Σ, corresponding to the action, u, in E, since no action appears more than once in E. Every pair of transitions t_1, t_2 arising from atomic actions in E_1 has the same connectivity, *i.e.* $t_1 \bowtie t_2$. Similarly, for each choice subexpression of E_2 and E_3. The set of transitions, X, is given by $X_1 \cup X_2$, where:

$$
\begin{aligned}
X_1 &= \{\{t(\widehat{u_1}), t(u_i)\} \mid 1 \leq i \leq n\} \\
X_2 &= \{\{t(\widehat{u_1}), t(c_{i_1}\} \mid 1 \leq i \leq m\}
\end{aligned}
$$

Every transition in X, individually, is a valid synchronisation. However, all the transitions can be represented by the addition of synchronisation operations to E if and only if there is a satisfying assignment to the instance of ONE-IN-THREE 3SAT. The set of transitions, X, is designed to enforce choices between equivalent synchronisations. The form of the subexpressions, E_1, E_2 and E_3, ensure that the choices between equivalent synchronisations correspond to choices of assignment to literals in the instance of ONE-IN-THREE 3SAT.

- E_1 provides the conjugates of the set of literals. Every synchronisation in X contains exactly one transition arising from E_1. In the implementation of E_1, every transition has the same connectivity. Hence, for each synchronisation $x \in X$, no constraints on the choice of action for the synchronisation operation used to create x are imposed by the transitions arising from E_1.

- E_2 represents the set of clauses, C, of the instance of ONE-IN-THREE 3SAT. Each clause is represented by a choice construct. There is one synchronisation between each choice construct, and E_1, enforcing the choice of exactly one of the three literals in the clause.

- E_3 is used to enforce the choice between each literal and its negation (*e.g.* a and $\neg a$). There is one synchronisation between each choice construct in E_3, and E_1. The form of E means that the choice between a literal and its negation must be maintained throughout the set of clauses, C, because any synchronisation operation used to create a transition in the set X_1 (synchronisations of transitions arising from E_1 and E_3) must be in the scope of E_1, E_3, and therefore also E_2.

The implementation, Σ, of E can be constructed in polynomial time. Each new transition, $x \in X$ has the form $\{t_1, t_2\}$, where t_1 and t_2 are transitions in Σ.

As an example, the transformation from ONE-IN-THREE 3SAT to SYNCHRONISATION ASSIGNMENT is illustrated using the instance of ONE-IN-THREE 3SAT given by:

$$\begin{aligned} U &= \{a, b, c\} \\ C &= \{\{a, b, \neg a\}, \{\neg b, c, \neg a\}, \{b, \neg a, \neg c\}\} \end{aligned}$$

The corresponding instance of SYNCHRONISATION ASSIGNMENT is shown in Figure 4.10. The transitions in X are indicated using dotted arcs. The expression, E is given by:

$$\begin{aligned} E &= (\widehat{a} \,[]\, \widehat{\neg a} \,[]\, \widehat{b} \,[]\, \widehat{\neg b} \,[]\, \widehat{c} \,[]\, \widehat{\neg c}) \\ &\;; \quad (a \,[]\, b \,[]\, \neg a); (\neg b \,[]\, c \,[]\, \neg a); (b \,[]\, \neg a \,[]\, \neg c) \\ &\;; \quad (a \,[]\, \neg a); (b \,[]\, \neg b); (c \,[]\, \neg c) \end{aligned}$$

The net in Figure 4.10 is an implementation of E **sy** $\{a, \neg b, \neg c\}$. This corresponds to a satisfying assignment, $a = true, b = false, c = false$ to the instance of ONE-IN-THREE 3SAT. In this example, there are no other possibilities for the synchronisation set, and hence no other satisfying assignments. There is some flexibility in the position of the synchronisation operators. However, the form of E sufficiently constrains the allowed positions for the **sy** operators to ensure that any valid synchronisation assignment corresponds to a satisfying assignment of the instance of ONE-IN-THREE 3SAT. In addition, no transition arising from E can be obtained as the result of a synchronisation operation, and no synchronised transition from X can be represented using the basic syntax.

The NP hardness result for SYNCHRONISATION ASSIGNMENT does not immediately imply that the synthesis of synchronisation is NP hard, because it is assumed that the input to the synthesis algorithm is an implementation of a Petri box. Instances of ONE-IN-THREE 3SAT for which there is no satisfying assignment are transformed into nets that are not the implementation of any box expression. Fortunately, a simple argument can be used to show that any

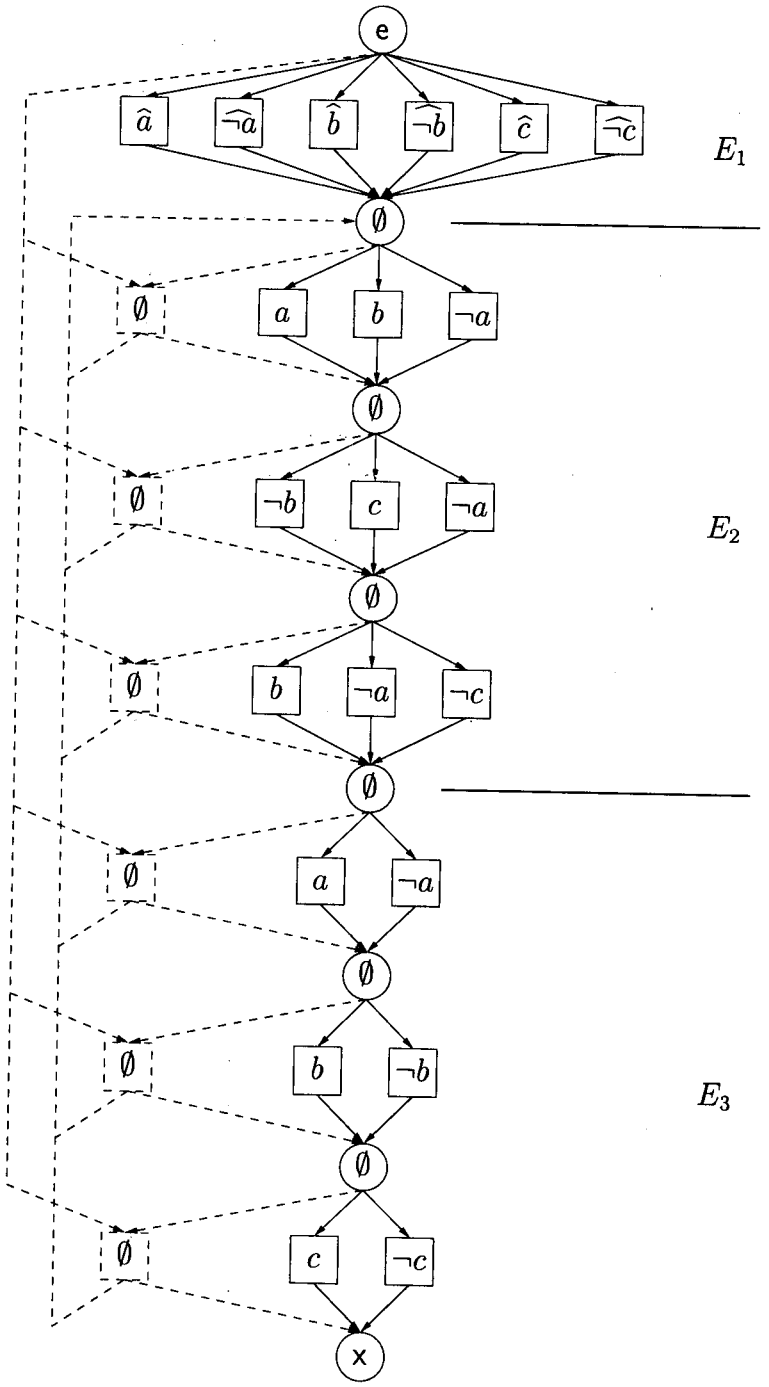


Figure 4.10: Transformation from an instance of ONE-IN-THREE 3SAT

efficient solution to BOX EXPRESSION SYNTHESIS can be extended to provide a solution to ONE-IN-THREE 3SAT.

Suppose there is an efficient algorithm for BOX EXPRESSION SYNTHESIS. The behaviour of this algorithm is undefined on an input that is not the implementation of a box expression: An incorrect result may be obtained, or the algorithm may not terminate at all. For any polynomial time algorithm, there exists fixed values c, d such that for an input of size n, the algorithm terminates in at most $c \cdot n^d$ steps. Thus, the case of non-termination can be detected in polynomial time by maintaining a count of the number of steps made by the algorithm. As soon as the upper bound has been exceeded, it is known that the input was not an implementation of a box expression.

Given an arbitrary instance of ONE-IN-THREE 3SAT the corresponding instance of SYNCHRONISATION ASSIGNMENT can be constructed in polynomial time, and passed as input to BOX EXPRESSION SYNTHESIS. The synthesis algorithm can be modified to recognise a non-termination condition, in which case it is known that there is no satisfying assignment for the instance of ONE-IN-THREE 3SAT. If the algorithm terminates with a synthesised expression, then a candidate satisfying assignment for the instance of ONE-IN-THREE 3SAT has been found. The candidate assignment can be checked in polynomial time. If the assignment does not satisfy the instance of ONE-IN-THREE 3SAT then the input to the synthesis algorithm cannot be an implementation of a box expression - therefore there is no satisfying assignment. Hence, an efficient solution to BOX EXPRESSION SYNTHESIS can provide an efficient solution to ONE-IN-THREE 3SAT. Therefore, under the assumption that $P \neq NP$, there is no efficient algorithm for BOX EXPRESSION SYNTHESIS.

Proposition 22 *The synthesis problem,* BOX EXPRESSION SYNTHESIS, *as defined in Section 4.1 is NP-hard.*

Proof: Follows from the argument and construction given above. □

4.2.5 Tractable solutions to synthesis with synchronisation

In this section, some possible approaches to dealing with the NP hardness result of Section 4.2.4 are discussed. It is possible that any of these approaches could be used to produce a satisfactory solution to the synthesis problem.

- **Restrict the input language:** The class of nets allowed as input to the synthesis algorithm could be restricted to be implementations of expressions derived from some subset of the syntax in Table 4.1. For example, if the use of the synchronisation operator is forbidden, the synthesis problem becomes tractable, as demonstrated in Chapter 3. A less severe restriction is to restrict the choice operator so that no atomic actions are allowed to appear in a choice context – *i.e.* all choice expressions have the form:

$$E_1 \,\square\, E_2 \,\square\, \ldots \,\square\, E_n$$

  where no E_i, for $1 \leq i \leq n$, is an atomic action. This certainly prevents the form of expression constructed by the transformation from ONE-IN-THREE 3SAT, and more generally, significantly reduces the scope for equivalent synchronisations. Another restriction that, intuitively, makes the problem simpler is to restrict atomic actions to be single basic actions, or sets of basic actions, instead of multisets of basic actions. However, the NP hardness result is based on expressions that contain only single basic actions as atomic actions. Of course, there is the possibility that the synthesis problem will be even more difficult when multisets of basic actions are permitted.

- **Heuristic solution:** Heuristics are generally more suited to optimisation problems where there is a range of acceptable solutions. In such cases, an algorithm may be efficient, but does not guarantee to give the best possible solution. Instead, heuristics are used to give a "good"

solution most of the time. The synthesis problem, however, is not an optimisation problem, as the result must be an expression whose implementation is isomorphic to the input net. In cases where there is more than one possible expression, every solution is regarded as equally good. Hence, any heuristic algorithm for the synthesis of synchronisation will take an exponential amount of time for some inputs, and therefore cannot be regarded as a tractable solution. The aim of the heuristics are to synthesise an expression quickly, for the majority of input nets.

- **Increase expressiveness of output language:** For certain problems, increasing the size of the solution space eliminates the constraints which make the problem hard. For example, INTEGER PROGRAMMING is NP-complete [33, 15, 25]. However, if the solution is allowed to be a set of real numbers rather than a set of integers, the problem becomes LINEAR PROGRAMMING [34] which has a polynomial time solution. For the synthesis algorithm, increasing the size of the solution space involves adding new operators to the syntax used to represent the synthesised expression, or replacing existing operators with more expressive ones. A new operator could be designed specifically to cope with the NP-hardness result. Alternatively, an additional operator from the box calculus could be added to the syntax of Table 4.1. Using an existing operator is more desirable because there is already justification for its inclusion in the box calculus. Two such operators are refinement and scoping. The refinement operator allows subexpressions to be moved outside the scope of a synchronisation operator. For example, using the refinement operator, the net in Figure 4.6 can be synthesised to:

$$(((X \,\square\, a) \parallel \widehat{a})\ \mathbf{sy}\ a)[X \leftarrow \widehat{a}]$$

In addition, an expression, E_r, for the net in Figure 4.10 can be synthesised without having to find a satisfying assignment to the corresponding

instance of ONE-IN-THREE 3SAT:

$$\begin{aligned} E_r \;=\; & (((\hat{a} \,[]\, \widehat{\neg a} \,[]\, \hat{b} \,[]\, \widehat{\neg b} \,[]\, \hat{c} \,[]\, \widehat{\neg c}) \\ & ;\; (X_1 \,[]\, \neg a); (X_2 \,[]\, \neg a); (X_3 \,[]\, \neg a) \\ & ;\; (a \,[]\, \neg a); (b \,[]\, \neg b); (c \,[]\, \neg c)) \textbf{ sy } \{\neg a, b, \neg c\}) \\ & [X_1 \leftarrow a \,[]\, b][X_2 \leftarrow \neg b \,[]\, c][X_3 \leftarrow b \,[]\, \neg c] \end{aligned}$$

With the introduction of new basic actions not used elsewhere, the scoping operator can be used to represent each synchronised transition independently of every other synchronised transition. For example, using the scoping operator, and a new basic action, n, the net in Figure 4.6 can be synthesised to:

$$[n : (\hat{a} \,[]\, a \,[]\, n) \;||\; (\hat{a} \,[]\, \hat{n})]$$

As with refinement, the scoping operator allows an expression, E_s, for the net in Figure 4.10 to be synthesised without entailing the production of a satisfying assignment to the corresponding instance of ONE-IN-THREE 3SAT:

$$\begin{aligned} E_s \;=\; & [\{n_1, n_2, n_3, n_4, n_5, n_6\} : (\hat{a} \,[]\, \widehat{\neg a} \,[]\, \hat{b} \,[]\, \widehat{\neg b} \,[]\, \hat{c} \,[]\, \widehat{\neg c} \,[]\, \widehat{n_1} \,[]\, \widehat{n_2} \,[] \\ & \widehat{n_3} \,[]\, \widehat{n_4} \,[]\, \widehat{n_5} \,[]\, \widehat{n_6}) \\ & ;\; (a \,[]\, b \,[]\, \neg a \,[]\, n_1); (\neg b \,[]\, c \,[]\, \neg a \,[]\, n_2); (b \,[]\, \neg a \,[]\, \neg c \,[]\, n_3) \\ & ;\; (a \,[]\, \neg a \,[]\, n_4); (b \,[]\, \neg b \,[]\, n_5); (c \,[]\, \neg c \,[]\, n_6)] \end{aligned}$$

The scoping operator is used in place of the synchronisation operator, rather than in addition to it, as with refinement.

- **Use a different notion of equivalence:** An alternative structural equivalence to isomorphism could be used as a basis for equality of Petri boxes. For example, if duplication equivalence is used, then it is possible to produce multiple copies of each synchronised transition. Hence, the

net in Figure 4.10 can be synthesised to:

$$\begin{aligned} E_d \;=\; & ((\widehat{a} \,\Box\, \widehat{\neg a} \,\Box\, \widehat{b} \,\Box\, \widehat{\neg b} \,\Box\, \widehat{c} \,\Box\, \widehat{\neg c}) \\ ; \;\; & (a \,\Box\, b \,\Box\, \neg a); (\neg b \,\Box\, c \,\Box\, \neg a); (b \,\Box\, \neg a \,\Box\, \neg c) \\ ; \;\; & (a \,\Box\, \neg a); (b \,\Box\, \neg b); (c \,\Box\, \neg c)) \;\; \mathbf{sy} \;\; \{a, \neg a, b, \neg b, c, \neg c\} \end{aligned}$$

The implementation of E_d is duplication equivalent to the net in Figure 4.10, although not necessarily isomorphic to it.

$E ::=$	α	Atomic action
	$E \parallel E$	Parallel composition
	$E \,\Box\, E$	Choice composition
	$E; E$	Sequential composition
	$[E * E * E]$	Iteration
	$[A : E]$	Scoping

Table 4.3: Output box expression syntax

Of the possible approaches described above, the extension of the output syntax by replacing the synchronisation operator with the scoping operator was chosen to be investigated further. The syntax used to represent synthesised expressions is given in Table 4.3. The use of the scoping operator has several desirable properties:

- The overlap between the basic syntax, and the scoping operator is such that transitions that can be represented using the scoping operator can be identified by a local analysis of the net. Identifying transitions arising from the synchronisation operator required a global analysis of the net, as shown by the example in Figure 4.6.

- The scoping operator allows synchronised transitions to be represented independently of each other. In particular, this resolves the choice between equivalent synchronisations, and eliminates the constraints that the NP-hardness result relied upon.

- To represent synchronised transitions using the scoping operator, new basic actions are introduced. These basic actions do not appear in the labels of the transitions of the input net. Hence, there is a reasonable degree of flexibility in the positioning of the scoping operators. Placing the scoping operators so that they enclose as large a subexpression as possible, means that the possible positions are limited to:

  - The top level of the expression.
  - Immediately inside an iteration operator.

- It is possible to represent multi-way synchronisations using the scoping operator. For example, the net in Figure 4.2 can be synthesised to $[N : E]$, where:

$$
\begin{aligned}
N &= \{n_4, n_5, n_6, n_7, n_8, n_9, n_{10}, n_{11}\} \\
E &= (\{a, a\} \,\square\, \{\widehat{n_5}\} \,\square\, \{n_4, a, b\} \,\square\, \{n_6, n_7, b, b\} \,\square\, \{\widehat{n_8}\} \,\square\, \{\widehat{n_{11}}\}) \\
&\parallel (\{\widehat{a}, b\} \,\square\, \{\widehat{n_4}\} \,\square\, \{\widehat{n_6}\} \,\square\, \{\widehat{n_7}\} \,\square\, \{\widehat{n_{10}}\}) \\
&\parallel (\{\widehat{a}, c\} \,\square\, \{c, c, n_8, n_9\} \,\square\, \{b, c, n_{10}, n_{11}\} \,\square\, \{a, c, n_5\})
\end{aligned}
$$

  Although the synthesised expression is much larger than the original expression, it has a regular structure. This is illustrated by Figure 4.11, which shows the implementation of E (*i.e.* before the application of the scoping operator).

The technique of representing synchronised transitions using the scoping operator cannot be applied to infinite synchronisations. However, infinite nets are not considered as input to the synthesis algorithm. The synthesis problem is restated with the modification to represent the synthesised expression using the scoping operator. It is this definition of the problem that is used for the remainder of the chapter.

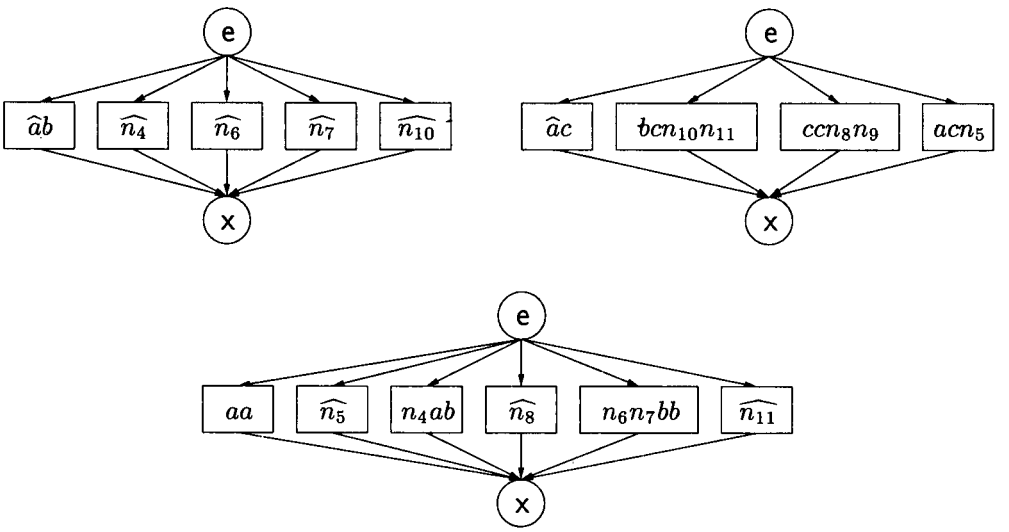


Figure 4.11: Synthesis of multi-way synchronisation using the scoping operator

BOX EXPRESSION SYNTHESIS

INSTANCE: Net, Σ, member of the class of Petri boxes allowed as input.

SOLUTION: Box expression, E from the syntax in Table 4.3,

such that $\mathsf{box}(E) = [\Sigma]$.

4.3 The synthesis algorithm

BOX EXPRESSION SYNTHESIS takes as input a net, Σ, which is the implementation of some unknown box expression from the syntax in Table 4.1. The output of the algorithm is a box expression, E, from the syntax in Table 4.3, such that Σ is an implementation of E.

In this section, an algorithm for BOX EXPRESSION SYNTHESIS, based on the CANONICAL BOX EXPRESSION SYNTHESIS algorithm of Chapter 3, is presented. A new synthesis rule, SCOPING is introduced. This rule is described in detail in Section 4.4.

4.3.1 Outline of the algorithm

Let Σ be an implementation of an expression from the syntax in Table 4.1. The underlying net, Σ_a, of Σ is obtained by removing the set of transitions, $T_{sc}(\Sigma)$:

$$\Sigma_a = \Sigma \ominus T_{sc}(\Sigma)$$

Theorem 3 in Section 4.5 shows that Σ_a is the implementation of a basic syntax box expression. Hence, a canonical form expression, E_a, can be synthesised from the underlying net, Σ_a, by the CANONICAL BOX EXPRESSION SYNTHESIS algorithm of Chapter 3. E_a is known as the underlying expression for Σ.

The pseudo-code for BOX EXPRESSION SYNTHESIS is given below. The root node, N, is initialised with the underlying net of the input net, Σ. The SYNTHESISE procedure, described in Section 4.3.4, finds the underlying expression for Σ, and constructs the equivalence classes of the relation $\sim_\phi$. PRUNE discards those parts of the expression tree constructed by SYNTHESISE, whose purpose was purely for the computation of the equivalence classes of $\sim_\phi$. The pruned expression tree is similar to the one that would be obtained using CANONICAL BOX EXPRESSION SYNTHESIS. The SCOPING rule deals with the set of transitions, $T_{sc}(\Sigma)$, by augmenting the underlying expression with applications of the scoping operator. The SCOPING synthesis rule is described in Section 4.4.

BOX EXPRESSION SYNTHESIS(Σ)

1 N=*new* node
2 N.net=$\Sigma \ominus T_{sc}$
3 SYNTHESISE(N)
4 PRUNE(N)
5 SCOPING(N, Σ)
6 **return** EXPRESSION(N)

The remainder of this section is concerned with the modifications to the CANONICAL BOX EXPRESSION SYNTHESIS algorithm to compute the equivalence classes of the relation $\sim_\phi$. These equivalence classes partition the set of transitions in the underlying net of Σ, and are used by the SCOPING synthesis rule.

4.3.2 Data structure

The synthesis algorithm constructs a tree data structure which represents the synthesised expression. Each node of the tree has the form shown in Figure 4.12. The net field contains the net to be synthesised. The synthesis algorithm analyses the net to determine the synthesis rule to be applied, and sets the type field accordingly. If the type is atomic action, then the node is a leaf node, and the action field is set. Otherwise, the node is internal, and the list field is used to store an ordered list of subnets obtained by the net decomposition performed by the synthesis rule.

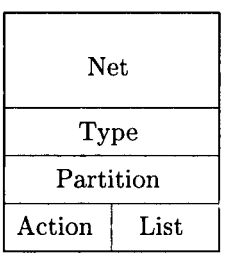


Figure 4.12: Data structure of a Node

The data structure has been modified from Chapter 3 to include the partition field. This field is an ordered list of sets of the transitions contained in the net field, and represents the partitioning of the transitions corresponding to the equivalence relation $\sim_\phi$. The mapping between actions in the expression and sets of transitions is implicit in the ordering of the list. The order atomic

action nodes are reached using a depth-first traversal of the tree match the ordering of the sets of transitions in the list.

4.3.3 Modified synthesis rules

Extended versions of the atomic action and iteration synthesis rules are described in this section. The extension from the original rules, described in Chapter 3, are to allow the transitions in the input net to be partitioned according to the equivalence relation, $\sim_\phi$.

Atomic action

The atomic action synthesis rule is applied when the input net contains a single transition, t. The partition field of the node is initialised to contain a single set $\{t\}$. Figure 4.13 shows the effect of the atomic action synthesis rule on the implementation of an expression $E = a$.

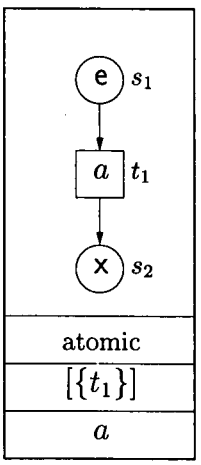


Figure 4.13: Atomic action synthesis rule

Iteration

Let Σ be a implementation of a basic syntax box expression which satisfies the preconditions of the iteration synthesis rule (see Chapter 3). The itera-

tion synthesis rule performs a partial decomposition of Σ, resulting in a net containing two connected components, Σ_{c_1} and Σ_{c_2} which are isomorphic to each other (shown in Chapter 3). For example, for an implementation of the expression $E = [a*b*c]$, the partial decomposition shown in Figure 4.14 would be performed.

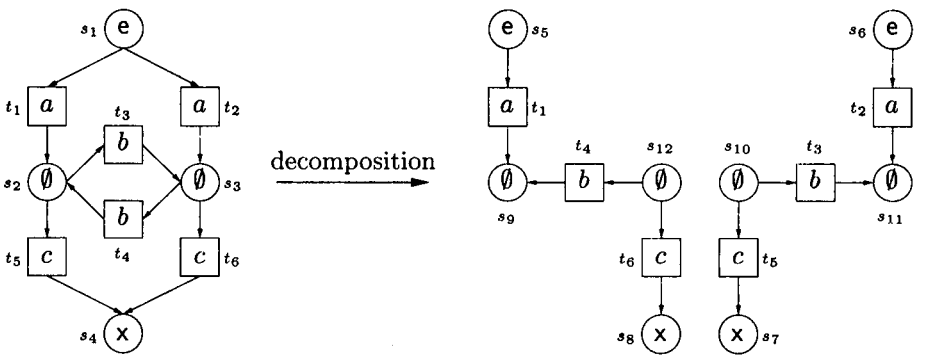


Figure 4.14: Decomposition performed by iteration synthesis rule

Instead of discarding one component, and decomposing the other component into three subnets, corresponding to the subexpressions E_1, E_2 and E_3 in $[E_1 * E_2 * E_3]$, the modified iteration synthesis rule decomposes both Σ_{c1} and Σ_{c2}. Hence, six subnets, $\Sigma_1, ..., \Sigma_6$, are produced by the net decomposition performed by the iteration synthesis rule. Let E_i, for $1 \leq i \leq 6$, be the canonical form expression synthesised from Σ_i. Since the two components in the partial decomposition are isomorphic, the expression $[E_1 * E_2 * E_3]$ will be identical to $[E_4 * E_5 * E_6]$. Hence, an isomorphism between the transitions in Σ_{c_1} and Σ_{c_2} can be obtained, allowing the transitions of Σ to be partitioned according to the relation, $\sim_\phi$.

For example, for an implementation of the expression, $E = [a * b * c]$, the decomposition shown in Figure 4.15 will be obtained. The method for computing the partition field of the root node in Figure 4.15 will be explained in Section 4.3.4.

Once the entire expression tree and the equivalence classes of the relation $\sim_\phi$ have been constructed, the PRUNE function discards the second set of

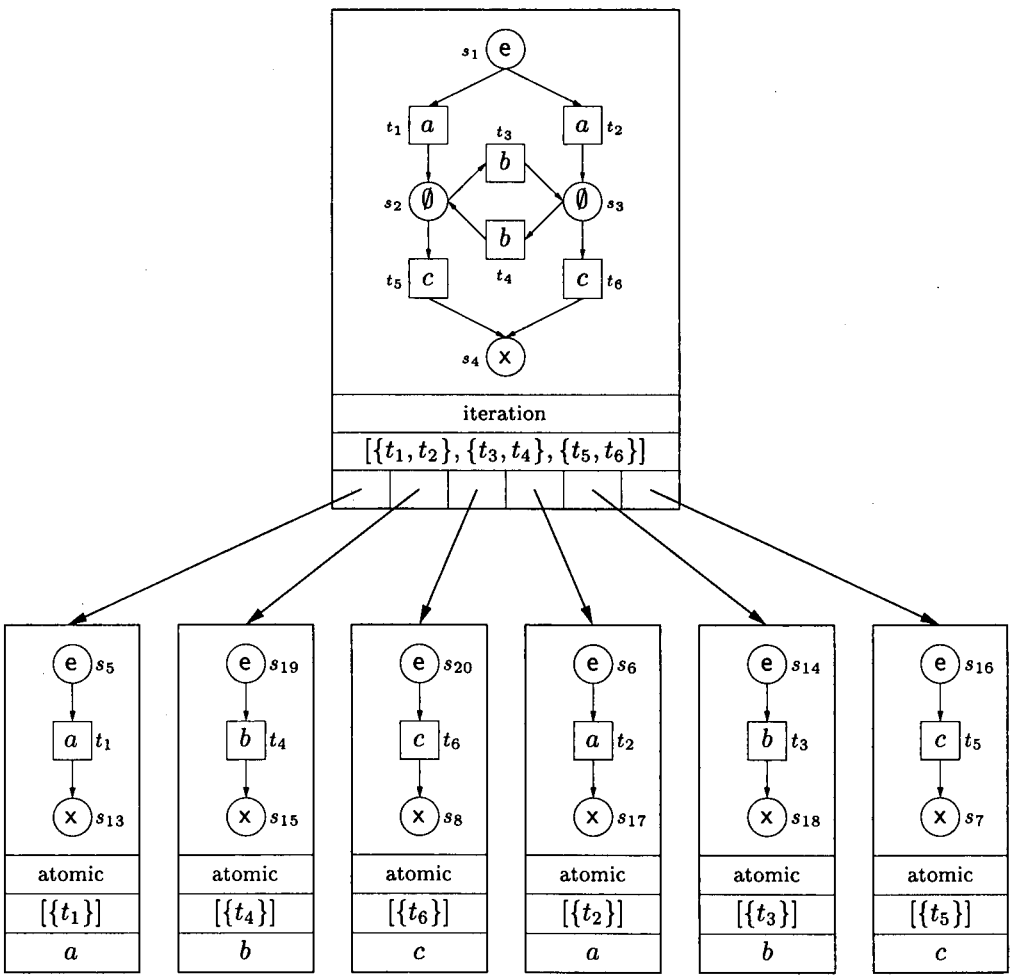


Figure 4.15: Iteration synthesis rule

three child subtrees of each iteration node. Hence, the modifications described here have no effect on the output of the algorithm. The analysis of the time complexity in Chapter 3 assumes that no portion of the input net is discarded during the synthesis process. Therefore, the time complexity of the algorithm (not taking into account the time complexity of SCOPING), remains at $O(n^5)$, where n is the number of nodes in the input net.

4.3.4 Partitioning the transitions

The pseudo-code below gives the SYNTHESISE procedure called by BOX EXPRESSION SYNTHESIS. This code is the ORDERED SYNTHESISE procedure, presented in Chapter 3, extended by lines 12 and 13 to compute the partition field of the node data structure. The remainder of the synthesis algorithm, including the ANALYSE function, and the synthesis rules PARALLEL, CHOICE and SEQUENCE is exactly as described in Chapter 3, and is not repeated here. The ATOMIC and ITERATION synthesis rules are as described in Chapter 3, with the modifications of Section 4.3.3. The sorting of the list field of the node, performed in line 11, is based on a total order of box expressions, such as that used in Section 3.5.3 in Chapter 3.

```
SYNTHESISE(N)
1    N.type=ANALYSE(N.net)
2    case N.type
3         atomic: ATOMIC(N)
4         parallel: PARALLEL(N)
5         choice: CHOICE(N)
6         iteration: ITERATION(N)
7         sequence: SEQUENCE(N)
8    for each node N' in N.list
9        do SYNTHESISE(N')
```

```
10  if N.type=parallel or choice
11     then sort(N.list)
12  if N.type≠atomic
13     then N.partition = PARTITION(N)
```

The PARTITION function computes the partitioning of the transitions of the net, *N.net*, from the partition fields of the child nodes of N. Two operations on lists, + and $\cup$, are used in the computation. + is an append operation, with, for example:

$$[\{t_1, t_4\}, \{t_2\}] + [\{t_5, t_3\}, \{t_7, t_8\}] = [\{t_1, t_4\}, \{t_2\}, \{t_5, t_3\}, \{t_7, t_8\}]$$

The empty list is represented by ϵ, and for any list L, $L + \epsilon = L$. The list union operation, $\cup$, is defined for pairs of lists of equal length, and the result list has the same length as the operand lists. For lists $L_1 = [A_1, A_2, ..., A_n]$, and $L_2 = [B_1, B_2, ..., B_n]$, The list union, $L_1 \cup L_2$ is defined by:

$$L_1 \cup L_2 = [A_1 \cup B_1, ..., A_i \cup B_i, ..., A_n \cup B_n]$$

Hence, for example:

$$[\{t_1, t_4\}, \{t_2\}] \cup [\{t_5, t_3\}, \{t_7, t_8\}] = [\{t_1, t_4, t_5, t_3\}, \{t_2, t_7, t_8\}]$$

```
PARTITION(N)
1   X = [ ]
2   if N.type = iteration
3      then let φ1, ..., φ6 be the partition fields
                of the child nodes of N (in order)
4           for i = 1 to 3
5               X = X + (φi ∪ φi+3)
6      else for each node N' in N.list (in order)
7           X = X + N'.partition
8   return X
```

For example, the PARTITION function applied to the root node in Figure 4.15 returns the list:

$$[\{t_1, t_2\}, \{t_3, t_4\}, \{t_5, t_6\}]$$

which is constructed from the partition fields of the leaf nodes in Figure 4.15 as follows:

$$([\{t_1\}] \cup [\{t_2\}]) + ([\{t_4\}] \cup [\{t_3\}]) + ([\{t_6\}] \cup [\{t_5\}])$$

4.3.5 Example

In this section, an example input to the synthesis algorithm is introduced. The results of BOX EXPRESSION SYNTHESIS up to, but not including, the call to SCOPING are presented. The action of the scoping synthesis rule on this example is described in Section 4.4.

The net used as input to BOX EXPRESSION SYNTHESIS is shown in Figure 4.16. This net, Σ, is an implementation of the expression:

$$(d; [(d; ((\emptyset \,\square\, (\{a, a\} \parallel \{\widehat{a}, b\} \parallel \{\widehat{a}, c\}))\ \mathbf{sy}\ a); \widehat{d})\ \mathbf{sy}\ d * a * [c * b * \widehat{c}]\ \mathbf{sy}\ c])\ \mathbf{sy}\ d$$

Those transitions arising from synchronisation operations are indicated by dotted boxes and arcs in Figure 4.16. Since all the synchronisation operations are finite, Σ is a suitable input to the synthesis algorithm. An underlying net, Σ_a, unique up to isomorphism with $\Sigma - T_{sc}(\Sigma)$, is shown in Figure 4.17. Σ_a is used to initialise the *net* field of the node, N.

The call SYNTHESISE(N) constructs the tree data structure given in Figure 4.18. The net fields of the nodes are omitted in Figure 4.18. The partitions corresponding to the partition field entries, ϕ_1 to ϕ_8 are given in Table 4.4.

The call to PRUNE in Line 4 of BOX EXPRESSION SYNTHESIS results in the expression tree shown in Figure 4.19. The underlying expression, which can be obtained from the tree data structure, is:

$$d; [d; (\{a, a\} \parallel \{\widehat{a}, b\} \parallel \{\widehat{a}, c\}); \widehat{d} * a * [c * b * \widehat{c}]]$$

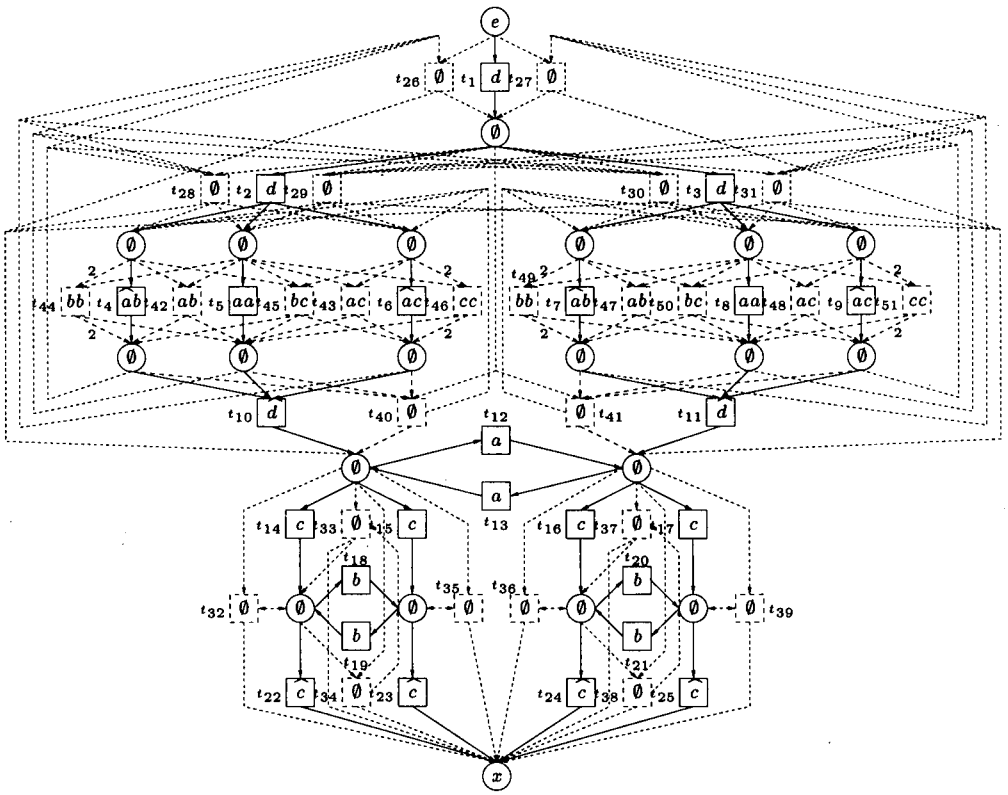


Figure 4.16: Input to the synthesis algorithm

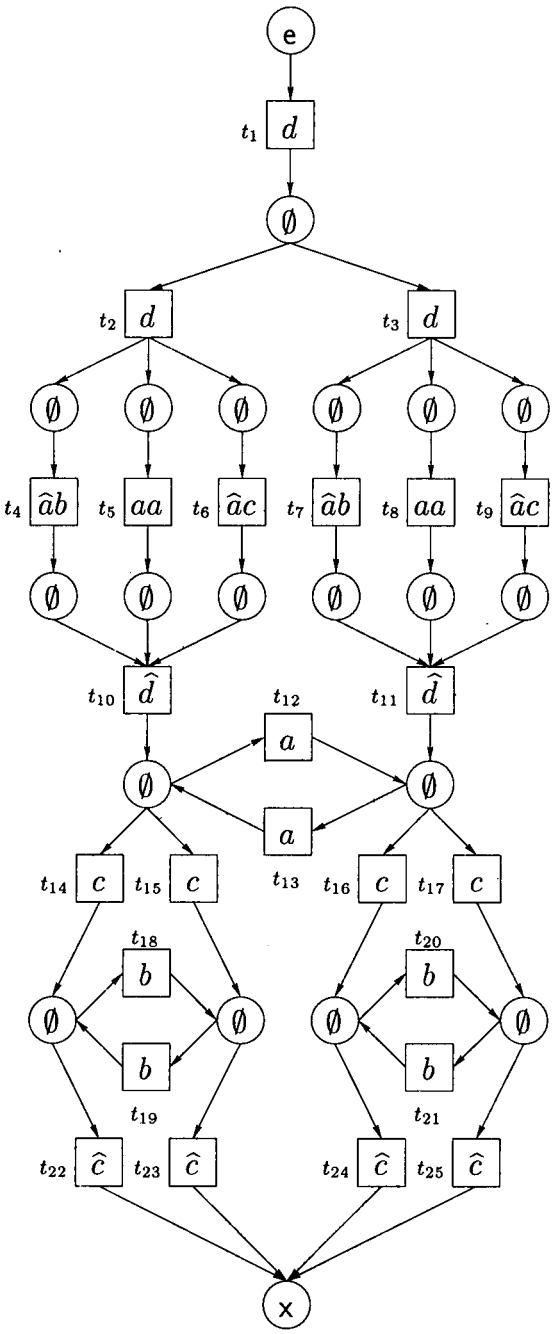


Figure 4.17: Underlying net

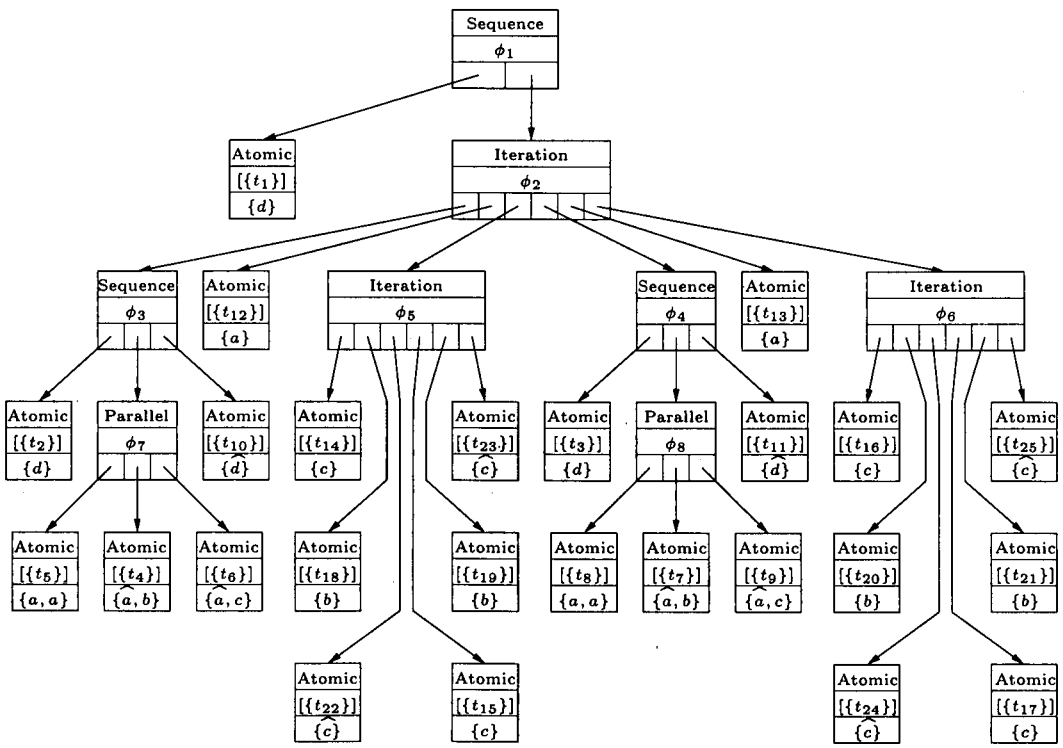


Figure 4.18: Tree structure constructed by SYNTHESISE

$$\phi_1 = [\{t_1\}, \{t_2, t_3\}, \{t_5, t_8\}, \{t_4, t_7\}, \{t_6, t_9\}, \{t_{10}, t_{11}\}, \{t_{12}, t_{13}\},$$
$$\{t_{14}, t_{15}, t_{16}, t_{17}\}, \{t_{18}, t_{19}, t_{20}, t_{21}\}, \{t_{22}, t_{23}, t_{24}, t_{25}\}]$$

$$\phi_2 = [\{t_2, t_3\}, \{t_5, t_8\}, \{t_4, t_7\}, \{t_6, t_9\}, \{t_{10}, t_{11}\}, \{t_{12}, t_{13}\},$$
$$\{t_{14}, t_{15}, t_{16}, t_{17}\}, \{t_{18}, t_{19}, t_{20}, t_{21}\}, \{t_{22}, t_{23}, t_{24}, t_{25}\}]$$

$$\phi_3 = [\{t_2\}, \{t_5\}, \{t_4\}, \{t_6\}, \{t_{10}\}]$$

$$\phi_4 = [\{t_3\}, \{t_8\}, \{t_7\}, \{t_9\}, \{t_{11}\}]$$

$$\phi_5 = [\{t_{14}, t_{15}\}, \{t_{18}, t_{19}\}, \{t_{22}, t_{23}\}]$$

$$\phi_6 = [\{t_{16}, t_{17}\}, \{t_{20}, t_{21}\}, \{t_{24}, t_{25}\}]$$

$$\phi_7 = [\{t_5\}, \{t_4\}, \{t_6\}]$$

$$\phi_8 = [\{t_8\}, \{t_7\}, \{t_9\}]$$

Table 4.4: Partitioning of the transitions in the underlying net

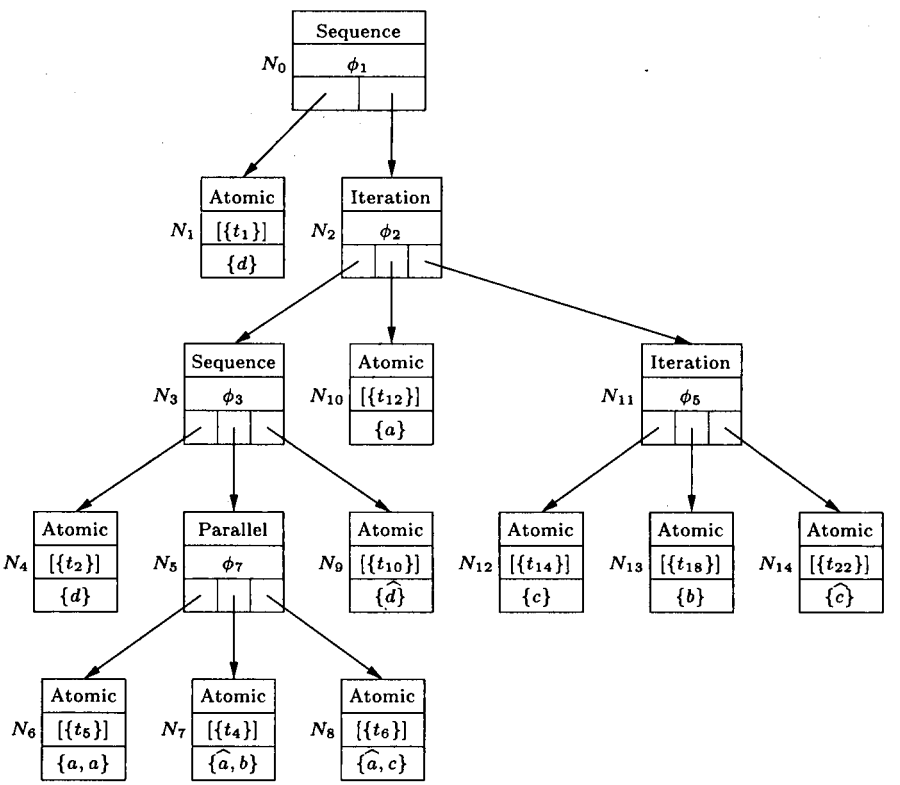


Figure 4.19: Pruned expression tree

4.4 Scoping synthesis rule

The scoping synthesis rule is applied after an expression tree for the underlying net, $\Sigma - T_{sc}(\Sigma)$, has been synthesised. The scoping rule inserts scoping operators, and atomic actions into expression tree in such a way that they generate exactly the set of transitions, $T_{sc}(\Sigma)$, that were removed from the input net. Central to the scoping synthesis rule is the relation between the transitions in the underlying net and the atomic actions in the synthesised expression. This relation is represented in the expression tree by the partition field of the nodes in the tree.

A set of new basic actions, $\{n_1, \widehat{n_1}, n_2, \widehat{n_2}, ...\}$, is assumed to be available. These actions do not appear anywhere in the labels of transitions in the input net. Every atomic action added to the expression tree by the scoping synthesis rule contains at least one new basic action, and each new basic action used, is scoped by one of the scoping operators inserted by the synthesis rule. The positions in the expression tree where the scoping operators are inserted are at the top level, and immediately inside each iteration operator.

The scoping synthesis rule, SCOPING deals with the set of transitions $T_{sc}(\Sigma)$, that were removed at the beginning of the synthesis process. SCOPING takes the pruned expression tree, N, synthesised from the underlying net, $\Sigma - T_{sc}(\Sigma)$, and the input net, Σ, and modifies N so that it represents an expression whose implementation is isomorphic to Σ. SCOPING begins by calling SCOPE to insert a scoping operator at the top level of the synthesised expression (*i.e.* at the root node, N, of the expression tree). At this point, the set of transitions to be dealt with is given by $T_{sc}(\Sigma)$. Tr is initialised to be the set of transitions remaining to be dealt with after the top level scoping operator has been added.

```
SCOPING(N, Σ)
1    Tr = SCOPE(N,T_sc(Σ))
2    VISIT(N,Tr)

VISIT(N,Tr)
```

```
1   if N.type=iteration
2     then for i=1 to 3
3               do Tr_1 = SCOPE(N.list[i],Tr ∩ N.list[i].partition)
4                  VISIT(N.list[i],Tr_1)
5   if N.type=choice or parallel or sequence
6     then for each node N' in N.list
7               VISIT(N',Tr ∩ N'.partition)
```

The recursive procedure, VISIT traverses the expression tree in a depth first fashion. The parameters of VISIT are N, the node currently being visited, and Tr, the set of transitions remaining to be dealt with in the current subtree (*i.e.* the subtree with root node N). When VISIT reaches a node whose type is iteration, SCOPE is called three times to modify the iteration expression from $[E_1 * E_2 * E_3]$ to $[[A_1 : E_1'] * [A_2 : E_2'] * [A_3 : E_3']]$. At each node, the set of transitions yet to be represented by a scoping operator is partitioned between the child nodes. For each $t \in Tr$, the nodes representing transitions from the set $T_b(t)$ must all be contained in the same subtree, otherwise they would have been represented by a higher level scoping operator. Hence, Tr can be partitioned using the partition field, ϕ, of each child node as follows:

$$Tr \cap \phi = \{t \in Tr \mid \forall t' \in T_b(t), \exists T' \in \phi : t' \in T'\}$$

A call to SCOPE(N, Tr) determines the subset of Tr that can be created by a scoping operator inserted into the expression tree at the node N. If the set of transitions that can be represented is non-empty, INSERT SCOPING is called to modify the expression tree. The pseudo code for SCOPE is given below.

```
SCOPE(N,Tr)
1   U = Tr
2   X = ∅
3   while U ≠ ∅
4         choose any t ∈ U
5         T_b' = φ(t_1) ⊙ φ(t_2) ⊙ ... ⊙ φ(t_n)
          where T_b(t) = {t_1, ..., t_n} and φ = N.partition
```

```
6        T' = a minimal set
             {t' ∈ T | (λ(t') = λ(t)) ∧ (∃A ∈ T'_b : T_b(t') = A)}
             such that ∀t', u ∈ T' : T_b(u) = T_b(t') ⇒ u = t'
7        if |T'| = |T'_b|
8          then X = X + {t}
9               Tr = Tr − T'
10       U = U − T'
11   if X ≠ ∅ then INSERT SCOPING(N,X)
12   return Tr
```

The variable U stores the transitions from Tr that still need to be checked. In Line 1, U is initialised to be Tr. The multiset X records a representative transition from Tr for each new basic action to be scoped at this level. The while loop (Lines 3-10) chooses an arbitrary transition, t, from U. $T_b(t)$ gives the unique set of base transitions that, collectively, have the same connectivity as t. For every base transition, there is a corresponding atomic action in the expression tree. The partition field of node N gives the mapping between the atomic actions of the subexpression, E', represented by the subtree with root N, and the transitions in the underlying net of the input net. Hence, any scoping operation on E' that synchronises the set of transitions $T_b(t) = \{t_1, ..., t_n\}$ will create the set of transitions T'_b, given by:

$$T'_b = \phi(t_1) \odot \phi(t_2) \odot ... \odot \phi(t_n)$$

where $\phi(t)$ is the equivalence class of $\sim_\phi$ containing t. The scoping operation represented by t can be inserted into the expression tree at the node N only if, for every set of base transitions in $A \in T'_b$, there is a transition t' in U with $T_b(t') = A$ and $\lambda(t') = \lambda(t)$. Hence, if the set of transitions, T' computed in Line 6 has the same size as T'_b, then t is a suitable representative for a scoping operation at this point in the expression tree, t can be added to the set X, and T' can be removed from Tr. The set of transitions T' will always contain at least one transition, as t satisfies the conditions for inclusion in T'. Hence, for each iteration of the while loop, U will decrease in size. If there are some transitions that can be created by a scoping operator at the

node N in the expression tree, then INSERT SCOPING is called. Let X be the multiset of transitions to represent using a scoping operation. Every transition in X can be represented independently of any other transition. This approach requires $O(|X|^2)$ new basic actions to be introduced. In some cases it is possible to impose a hierarchy (partial order) on X, based on the size of the set of base transitions for each $x \in X$. In such cases, only one new basic action is required for each $x \in X$, giving an upper bound on the number of new actions introduced equal to $|X|$.

```
INSERT SCOPING(N,X)
1   A = ∅
2   for each x, in X
3       do L = {(t, {n̂_i}) | t ∈ T_b(x) and each n_i is a distinct new action}
4           choose any pair (t, l) ∈ L
5           C = {n | (t', {n̂}) ∈ L ∧ t' ≠ t}
6           A = A ∪ C
7           replace (t, l) in L by (t, C + λ(x))
8           for each node N' (visited in depth-first order, starting at N)
9               if N'.type=atomic
10                then while ∃(t_a, l_a) ∈ L such that t' ∈N.partition(t_a)
                        where N'.partition=[{t'}]
11                          ADD(N', l_a)
12                          remove (t_a, l_a) from L
13  ADD SCOPING(N,A)
```

The set A, initialised in Line 1 of INSERT SCOPING is used to record the sets of new actions introduced for each $x \in X$. During each iteration of the main loop (Lines 2-12), the new actions used in that iteration are added to A in Line 6. Every new action is scoped by the node with type *scoping* inserted into the expression tree by Line 13.

Each transition, x in the multiset X is dealt with independently by the main loop. Lines 3-7 construct a set, L of pairs of transitions and actions (transition labels). The presence of an element (t, l) in L indicates that a transition with the same connectivity as t, and with the label l is required.

L is initialised in Line 3 to consist of entries $(t, \{\widehat{n_i}\})$, where $t \in T_b(x)$, is a member of the unique multiset of base transitions for the synchronised transition to be represented, and each n_i is a new basic action that has not already been used. One of the elements, (t, l) of L is chosen (at random) as distinguished, and the new basic action contained in l is discarded. The set C, constructed in Line 5, records the new actions introduced during the current iteration of the main loop, excluding the one discarded from the distinguished element of L. The new label assigned to the distinguished element of L in Line 7 is constructed from C, and the label of x. It is relatively simple to verify that scoping the set of transitions represented by L by the set of actions C results in a transition with the same connectivity and label as x.

Lines 8-12 traverse the expression tree in a depth-first manner, starting at N. When an atomic action node, N', which has a corresponding transition t_a such that $(t_a, l_a) \in L$, then ADD is called to create an atomic action node to represent (t_a, l_a). Once all of the entries in L have been dealt with, ADD SCOPING is called to ensure all the new actions are scoped.

ADD(N', l) is used to insert a new atomic action with label l in a choice context with an existing atomic action, represented by the node N' in the expression tree. Figure 4.20 shows the two ways in which the expression tree may be modified by a call to ADD(N', l). If the atomic action represented by N' is in a choice context (*i.e.* the parent node of N' has type choice), then the new atomic action is added to the existing choice context. If the existing atomic action is not part of a choice context, then a new choice node is inserted into the expression tree, as illustrated by (ii) in Figure 4.20. The new atomic action nodes inserted into the expression tree by ADD are given a partition field containing the empty partition, ϵ, indicating that the action was not originally part of the input net. Labelling the partition fields in this way has the property that the relationship between partition fields, given by PARTITION in Section 4.3.4, is preserved, since for any list X, $X + \epsilon = X$. Hence, for example, in Figure 4.20 (ii), the partition field of the new choice

node is defined to be $[\{t\}] + \epsilon$, which is $[\{t\}]$ – therefore the partition field of the sequence node remains as P.

ADD(N', l) is never called with N' equal to the root node of the expression tree – such a case could only arise if the synthesised expression is a single atomic action, α, which implies that any synchronised transitions must arise as a synchronisation between α and itself. However, it has been shown in Section 4.2 that synchronising a transition with itself produces an infinite synchronisation, and only finite nets are considered as input to the synthesis algorithm.

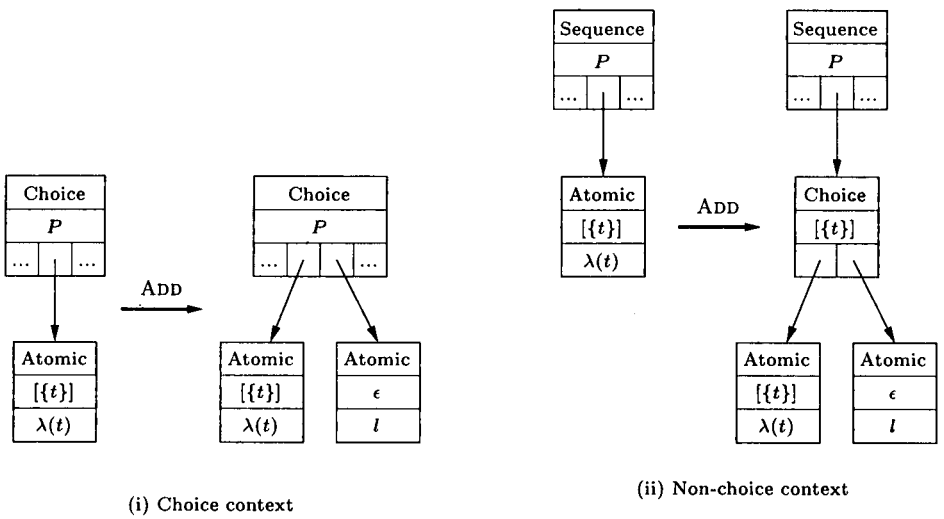


Figure 4.20: Adding actions to the expression tree

ADD SCOPING(N, A) is used to insert a new scoping operation at the point N in the expression tree. Figure 4.21 illustrates the modification made by ADD SCOPING. The new node, with type *scoping* inherits the parent of N, and the partition field of N. N is made a child of the new node. The scoping operation acts on the set of new basic actions introduced during the execution of INSERT SCOPING.

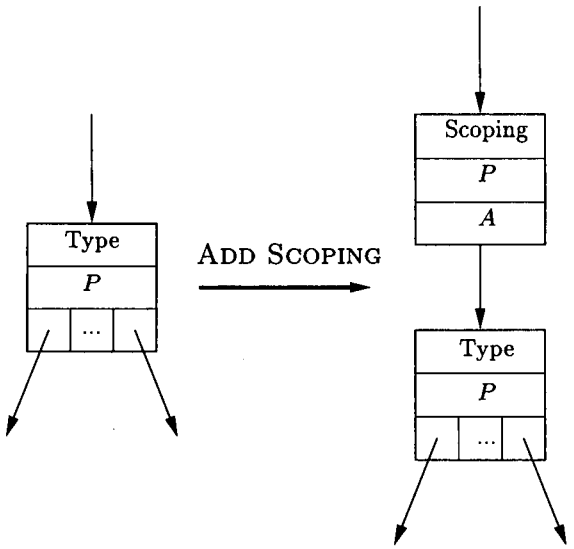


Figure 4.21: Adding a scoping operator to the expression tree

4.4.1 Example

In this section, the example introduced in Section 4.3.5 is continued. The net to be synthesised, Σ, is defined to be an implementation of the expression:

$$(d; [(d; ((\emptyset \,\square\, (\{a, a\} \parallel \{\widehat{a}, b\} \parallel \{\widehat{a}, c\}))\ \mathbf{sy}\ a); \widehat{d})\ \mathbf{sy}\ d * a * [c * b * \widehat{c}]\ \mathbf{sy}\ c])\ \mathbf{sy}\ d$$

The underlying net for Σ is shown in Figure 4.17, and the set of transitions, $T_{sc}(\Sigma)$, which are not included in Figure 4.17, are tabulated in Table 4.5. Each T_b entry in Table 4.5 consists of a multiset of base transitions that, collectively, have the same connectivity as the corresponding transition from $T_{sc}(\Sigma)$. For example, t_{44} inherits the connectivity of the multiset of transitions $\{t_4, t_4, t_5\}$. Hence, t_{44} has arcs of weight 2 that duplicate those of t_4, and also arcs of weight 1 duplicating those of t_5. Note that every base transition appears in Figure 4.17.

The scoping synthesis rule is called with SCOPING(N_0, Σ), where N_0 is the root node of the expression tree shown in Figure 4.19. Line 1 of SCOPING finds the subset of T_{sc} that can be represented by a scoping operation at the top level of the synthesised expression by calling SCOPE$(N_0, T_{sc}(\Sigma))$. Table 4.6

id	T_b	λ	id	T_b	λ	id	T_b	λ
t_{26}	$\{t_1, t_{10}\}$	$\emptyset$	t_{27}	$\{t_1, t_{11}\}$	$\emptyset$	t_{28}	$\{t_2, t_{10}\}$	$\emptyset$
t_{29}	$\{t_2, t_{11}\}$	$\emptyset$	t_{30}	$\{t_3, t_{10}\}$	$\emptyset$	t_{31}	$\{t_3, t_{11}\}$	$\emptyset$
t_{32}	$\{t_{14}, t_{22}\}$	$\emptyset$	t_{33}	$\{t_{14}, t_{23}\}$	$\emptyset$	t_{34}	$\{t_{15}, t_{22}\}$	$\emptyset$
t_{35}	$\{t_{15}, t_{23}\}$	$\emptyset$	t_{36}	$\{t_{16}, t_{24}\}$	$\emptyset$	t_{37}	$\{t_{16}, t_{25}\}$	$\emptyset$
t_{38}	$\{t_{17}, t_{24}\}$	$\emptyset$	t_{39}	$\{t_{17}, t_{25}\}$	$\emptyset$	t_{40}	$\{t_2, t_{10}\}$	$\emptyset$
t_{41}	$\{t_3, t_{11}\}$	$\emptyset$	t_{42}	$\{t_4, t_5\}$	$\{a, b\}$	t_{43}	$\{t_5, t_6\}$	$\{a, c\}$
t_{44}	$\{t_4, t_4, t_5\}$	$\{b, b\}$	t_{45}	$\{t_4, t_5, t_6\}$	$\{b, c\}$	t_{46}	$\{t_5, t_5, t_6\}$	$\{c, c\}$
t_{47}	$\{t_7, t_8\}$	$\{a, b\}$	t_{48}	$\{t_8, t_9\}$	$\{a, c\}$	t_{49}	$\{t_7, t_7, t_8\}$	$\{b, b\}$
t_{50}	$\{t_7, t_8, t_9\}$	$\{b, c\}$	t_{51}	$\{t_8, t_9, t_9\}$	$\{c, c\}$			

Table 4.5: Transitions to be represented by scoping

summarises the behaviour of SCOPE$(N_0, T_{sc}(\Sigma))$. The partition field of the root node, N_0 is:

$$\begin{aligned}\phi_1 \;=\; & [\{t_1\}, \{t_2, t_3\}, \{t_5, t_8\}, \{t_4, t_7\}, \{t_6, t_9\}, \{t_{10}, t_{11}\}, \{t_{12}, t_{13}\}, \\ & \{t_{14}, t_{15}, t_{16}, t_{17}\}, \{t_{18}, t_{19}, t_{20}, t_{21}\}, \{t_{22}, t_{23}, t_{24}, t_{25}\}]\end{aligned}$$

Hence, for example, the transitions, t_{18}, t_{19}, t_{20}, and t_{21} all arise from the same atomic action in the synthesised expression ($\{b\}$, represented by the node N_{13} in Figure 4.19).

The call SCOPE$(N_0, T_{sc}(\Sigma))$ performs 9 iterations of the while loop (Lines 3-10). Table 4.6 shows *no*, the number of the iteration of the while loop, the values of the variables U and X at the start of each iteration of the loop, t the transition chosen in Line 4, $T_b(t)$, the set of base transitions for t, the size of the set T'_b, calculated in Line 5, and T' the set of transitions constructed in Line 6. If $|T'| = |T'_b|$, as in iterations 1 and 2, then t is added to X, and the set T' is removed from the variable Tr. Hence Tr becomes $\{t_{28}, ..., t_{51}\}$ after the first iteration, and $\{t_{28}, t_{31}, ..., t_{39}, t_{42}, ..., t_{51}\}$ after the second and subsequent iterations. Once, U becomes $\emptyset$, at the beginning of iteration 10, the while loop

no	U	X	t	$\lambda(t)$	$T_b(t)$	$\lvert T'_b\rvert$	T'
1	$\{t_{26},...,t_{51}\}$	$\emptyset$	t_{26}	$\emptyset$	$\{t_1,t_{10}\}$	2	$\{t_{26},t_{27}\}$
2	$\{t_{28},...,t_{51}\}$	$\{t_{26}\}$	t_{28}	$\emptyset$	$\{t_2,t_{10}\}$	4	$\{t_{29},t_{30},t_{40},t_{41}\}$
3	$\{t_{28},t_{31},...,t_{39},t_{42},...,t_{51}\}$	$\{t_{26},t_{28}\}$	t_{28}	$\emptyset$	$\{t_2,t_{10}\}$	4	$\{t_{28},t_{31}\}$
4	$\{t_{32},...,t_{39},t_{42},...,t_{51}\}$	$\{t_{26},t_{28}\}$	t_{32}	$\emptyset$	$\{t_{14},t_{22}\}$	16	$\{t_{32},...,t_{39}\}$
5	$\{t_{42},...,t_{51}\}$	$\{t_{26},t_{28}\}$	t_{42}	$\{a,b\}$	$\{t_4,t_5\}$	4	$\{t_{42},t_{47}\}$
6	$\{t_{43},...,t_{46},t_{48},...,t_{51}\}$	$\{t_{26},t_{28}\}$	t_{43}	$\{a,c\}$	$\{t_5,t_6\}$	4	$\{t_{43},t_{48}\}$
7	$\{t_{44},t_{45},t_{46},t_{49},t_{50},t_{51}\}$	$\{t_{26},t_{28}\}$	t_{44}	$\{b,b\}$	$\{t_4,t_4,t_5\}$	6	$\{t_{44},t_{49}\}$
8	$\{t_{45},t_{46},t_{50},t_{51}\}$	$\{t_{26},t_{28}\}$	t_{45}	$\{b,c\}$	$\{t_4,t_5,t_6\}$	8	$\{t_{45},t_{50}\}$
9	$\{t_{46},t_{51}\}$	$\{t_{26},t_{28}\}$	t_{46}	$\{c,c\}$	$\{t_5,t_5,t_6\}$	6	$\{t_{46},t_{51}\}$
10	$\emptyset$	$\{t_{26},t_{28}\}$	–	–	–	–	–

Table 4.6: Call to SCOPE$(N_0, T_{sc}(\Sigma))$

terminates with $X = \{t_{26}, t_{28}\}$, and the call INSERT SCOPING$(N_0, \{t_{26}, t_{28}\})$ is made in Line 11.

x	$\lambda(x)$	$T_b(x)$	L	C
t_{26}	$\emptyset$	$\{t_1,t_{10}\}$	$\{(t_1,\{\widehat{n_1}\}),(t_{10},\{n_1\})\}$	$\{n_1\}$
t_{28}	$\emptyset$	$\{t_2,t_{10}\}$	$\{(t_2,\{\widehat{n_2}\}),(t_{10},\{n_2\})\}$	$\{n_2\}$

Table 4.7: Construction of L for INSERT SCOPING$(N_0, \{t_{26}, t_{28}\})$

Table 4.7 summarises the construction of L for the transitions t_{26} and t_{28}. There is a separate entry for each transition $x \in X$ (*i.e.* each iteration of the main loop). The column labelled $T_b(x)$ gives the multiset of base transitions used to construct the elements of L. The value given for L is that after Line 7, when the distinguished action has been chosen, and its label updated. The column labelled C contains the set of new basic actions introduced during each iteration of the main loop. There are many different, but equally valid possibilities for the value of L - the new basic actions, and the element chosen as distinguished could both be different. For example, $L = \{(t_1, \{n_{42}\}), (t_{10}, \{\widehat{n_{42}}\})\}$ is valid for $x = t_{26}$.

The calls ADD($N_1, \{\widehat{n_1}\}$) and ADD($N_9, \{n_1\}$) are made for the iteration of the main loop of INSERT SCOPING where $x = t_{26}$. The correspondence between nodes in the expression tree and base transitions can be seen in Figure 4.19. For example, N_1 and N_9 correspond to t_1 and t_{10} respectively. ADD($N_4, \{\widehat{n_2}\}$) and ADD($N_9, \{n_2\}$) are called for $x = t_{28}$. Finally, ADD SCOPING($N_0, \{n_1, n_2\}$) is called in Line 13 to scope the two new actions, n_1 and n_2.

The calls to ADD and ADD SCOPING modify the expression tree from that shown in Figure 4.19 to the one shown in Figure 4.22, representing the expression:

$$[\{n_1, n_2\} : (d \,\square\, \widehat{n_1}); [(d \,\square\, \widehat{n_2}); (\{a, a\} \parallel \{\widehat{a}, b\} \parallel \{\widehat{a}, c\}); (\widehat{d} \,\square\, n_1 \,\square\, n_2) * a * [c * b * \widehat{c}]]]$$

The call to SCOPE($N_0, T_{sc}(\Sigma)$) made in Line 1 of SCOPING returns with the set of remaining transitions $\{t_{28}, t_{31}, ..., t_{39}, t_{42}, ..., t_{51}\}$, which is used to initialise the variable Tr. VISIT(N_0, Tr), called in Line 2 of SCOPING traverses the expression tree in Figure 4.22 in depth-first order (hence the nodes N_0 to N_{14} are visited in order). When an iteration node is reached, three calls to SCOPE are made. For example, when node N_2 is visited, the following calls are made:

$$\text{SCOPE}(N_3, \{t_{28}, t_{42}, t_{43}, t_{44}, t_{45}, t_{46}\}) \tag{4.4}$$

$$\text{SCOPE}(N_{10}, \emptyset) \tag{4.5}$$

$$\text{SCOPE}(N_{11}, \{t_{32}, t_{33}, t_{34}, t_{35}\}) \tag{4.6}$$

since, for example, $Tr \cap N_3.partition = \{t_{28}, t_{42}, t_{43}, t_{44}, t_{45}, t_{46}\}$. Table 4.8 summarises the behaviour of (4.4), which results in a call to

$$\text{INSERT SCOPING}(N_3, \{t_{28}, t_{42}, t_{43}, t_{44}, t_{45}, t_{46}\})$$

Table 4.9 describes the construction of the set L for each transition to be represented by the scoping operator. The calls to ADD made from INSERT SCOPING are shown in Table 4.10. Line 13 of INSERT SCOPING calls ADD SCOPING($N_3, \{n_3, ..., n_{11}\}$), resulting in an expression tree representing the

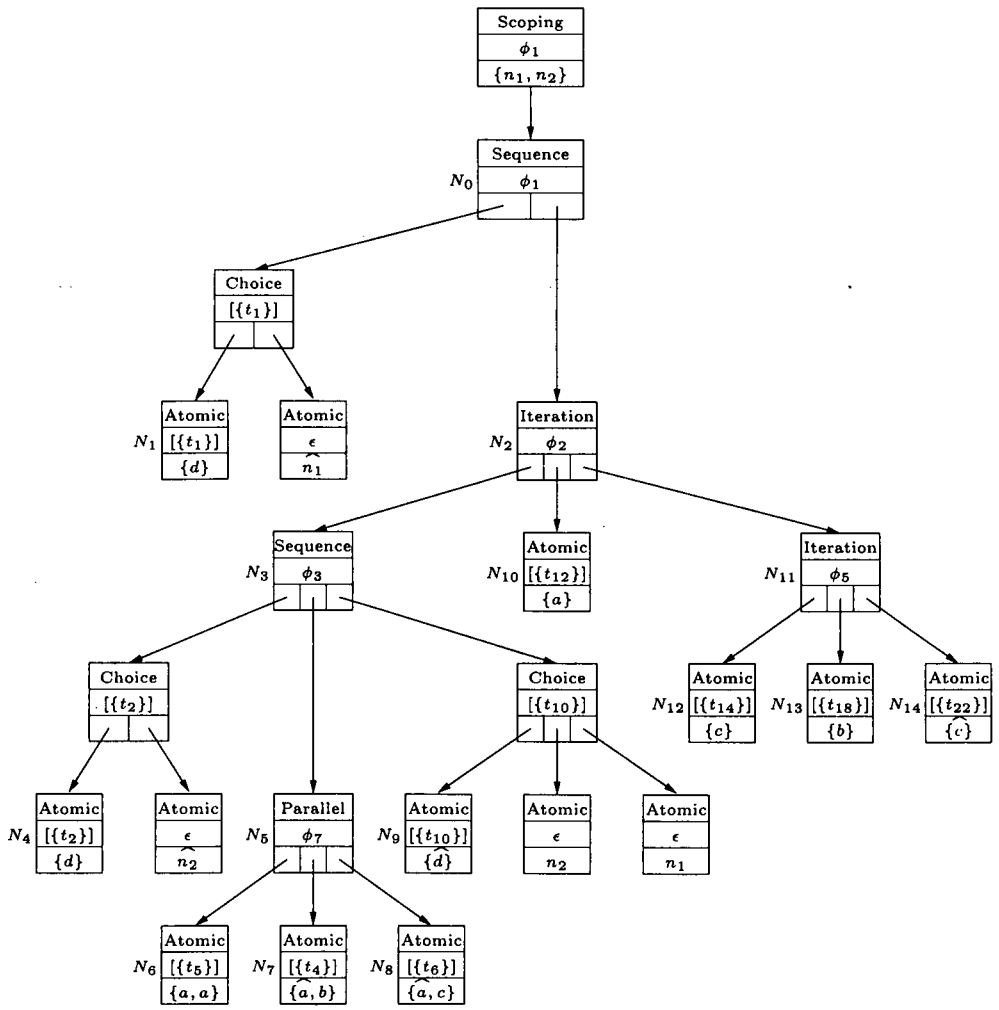


Figure 4.22: Scoped expression tree

no	U	X	t	$\lambda(t)$	$T_b(t)$	$\lvert T'_b\rvert$	T'
1	$\{t_{28}, t_{42}, ..., t_{46}\}$	$\emptyset$	t_{28}	$\emptyset$	$\{t_2, t_{10}\}$	1	$\{t_{28}\}$
2	$\{t_{42}, ..., t_{46}\}$	$\{t_{28}\}$	t_{42}	$\{a, b\}$	$\{t_4, t_5\}$	1	$\{t_{42}\}$
3	$\{t_{43}, ..., t_{46}\}$	$\{t_{28}, t_{42}\}$	t_{43}	$\{a, c\}$	$\{t_5, t_6\}$	1	$\{t_{43}\}$
4	$\{t_{44}, ..., t_{46}\}$	$\{t_{28}, t_{42}, t_{43}\}$	t_{44}	$\{b, b\}$	$\{t_4, t_4, t_5\}$	1	$\{t_{44}\}$
5	$\{t_{45}, t_{46}\}$	$\{t_{28}, t_{42}, ..., t_{44}\}$	t_{45}	$\{b, c\}$	$\{t_4, t_5, t_6\}$	1	$\{t_{45}\}$
6	$\{t_{46}\}$	$\{t_{28}, t_{42}, ..., t_{45}\}$	t_{46}	$\{c, c\}$	$\{t_5, t_5, t_6\}$	1	$\{t_{46}\}$
7	$\emptyset$	$\{t_{28}, t_{42}, ..., t_{46}\}$	–	–	–	–	–

Table 4.8: Call to SCOPE($N_3, \{t_{28}, t_{42}, t_{43}, t_{44}, t_{45}, t_{46}\}$)

expression:

$$E = [\{n_1, n_2\} : (d \,\square\, \widehat{n_1}); [[\{n_3, ..., n_{11}\} : E'] * a * [c * b * \hat{c}]]]$$

where $E' =$

$$(d \,\square\, \widehat{n_3} \,\square\, \widehat{n_2});$$
$$(\ (\{a, a\} \,\square\, \widehat{n_{11}} \,\square\, \widehat{n_{10}} \,\square\, \{\widehat{n_8}\} \,\square\, \{b, b, n_6, n_7\} \,\square\, \widehat{n_5} \,\square\, \{a, b, n_4\})$$
$$\|\ (\{\hat{a}, b\} \,\square\, \widehat{n_9} \,\square\, \widehat{n_7} \,\square\, \widehat{n_6} \,\square\, \widehat{n_4})$$
$$\|\ (\{\hat{a}, c\} \,\square\, \{c, c, n_{10}, n_{11}\} \,\square\, \{b, c, n_8, n_9\} \,\square\, \{a, c, n_5\})\ )$$
$$;(\hat{d} \,\square\, n_3 \,\square\, n_2 \,\square\, n_1)$$

x	$\lambda(x)$	$T_b(x)$	L	C
t_{28}	$\emptyset$	$\{t_2, t_{10}\}$	$\{(t_2, \{\widehat{n_3}\}), (t_{10}, \{n_3\})\}$	$\{n_3\}$
t_{42}	$\{a, b\}$	$\{t_4, t_5\}$	$\{(t_4, \{\widehat{n_4}\}), (t_5, \{a, b, n_4\})\}$	$\{n_4\}$
t_{43}	$\{a, c\}$	$\{t_5, t_6\}$	$\{(t_5, \{\widehat{n_5}\}), (t_6, \{a, c, n_5\})\}$	$\{n_5\}$
t_{44}	$\{b, b\}$	$\{t_4, t_4, t_5\}$	$\{(t_4, \{\widehat{n_6}\}), (t_4, \{\widehat{n_7}\}), (t_5, \{b, b, n_6, n_7\})\}$	$\{n_6, n_7\}$
t_{45}	$\{b, c\}$	$\{t_4, t_5, t_6\}$	$\{(t_4, \{\widehat{n_9}\}), (t_5, \{\widehat{n_8}\}), (t_6, \{b, c, n_8, n_9\})\}$	$\{n_8, n_9\}$
t_{46}	$\{c, c\}$	$\{t_5, t_5, t_6\}$	$\{(t_5, \{\widehat{n_{10}}\}), (t_5, \{\widehat{n_{11}}\}, (t_6, \{c, c, n_{10}, n_{11}\})\}$	$\{n_{10}, n_{11})\}$

Table 4.9: INSERT SCOPING ($N_3, \{t_{28}, t_{42}, t_{43}, t_{44}, t_{45}, t_{46}\}$) – Construction of L

x	*Calls*		
t_{28}	ADD$(N_4, \{\widehat{n_3}\})$	ADD$(N_9, \{n_3\})$	
t_{42}	ADD$(N_7, \{\widehat{n_4}\})$	ADD$(N_6, \{a, b, n_4\})$	
t_{43}	ADD$(N_6, \{\widehat{n_5}\})$	ADD$(N_8, \{a, c, n_5\})$	
t_{44}	ADD$(N_7, \{\widehat{n_6}\})$	ADD$(N_7, \{\widehat{n_7}\})$	ADD$(N_6, \{b, b, n_6, n_7\})$
t_{45}	ADD$(N_7, \{\widehat{n_9}\})$	ADD$(N_6, \{\widehat{n_8}\})$	ADD$(N_8, \{b, c, n_8, n_9\})$
t_{46}	ADD$(N_6, \{\widehat{n_{10}}\})$	ADD$(N_6, \{\widehat{n_{11}}\})$	ADD$(N_8, \{c, c, n_{10}, n_{11}\})$

Table 4.10: INSERT SCOPING $(N_3, \{t_{28}, t_{42}, t_{43}, t_{44}, t_{45}, t_{46}\})$ - Calls to ADD

The call to SCOPE$(N_{10}, \emptyset)$, (4.5), has no effect on the expression tree because there are no transitions from T_{sc} to be represented, and hence no need for an additional scoping operator.

The call to SCOPE$(N_{11}, \{t_{32}, t_{33}, t_{34}, t_{35}\})$ finds $X = \{t_{35}\}$[1], and hence calls INSERT SCOPING $(N_{11}, \{t_{35}\})$.

INSERT SCOPING constructs $L = \{(t_{15}, \{\widehat{n_{12}}\}), (t_{23}, \{n_{12}\})\}$, since $T_b(t_{35}) = \{t_{15}, t_{23}\}$. The nodes in the expression tree representing the atomic actions corresponding to the transitions t_{15} and t_{23} were removed by the call to PRUNE. However, these transitions are still present in $N_{11}.partition$, which has the value $[\{t_{14}, t_{15}\}, \{t_{18}, t_{19}\}, \{t_{22}, t_{23}\}]$. Therefore the nodes in the expression tree corresponding to $\{t_{15}\}$ and $\{t_{23}\}$ are respectively N_{12} and N_{14}, where N_{12} corresponds to the transition t_{14} and N_{14} corresponds to t_{22}. Hence the calls ADD$(N_{12}, \widehat{n_{12}})$ and ADD(N_{14}, n_{12}) are made. Finally, the call ADD SCOPING$(N_{11}, \{n_{12}\})$ is made, resulting in a tree representing the expression:

$$
\begin{aligned}
E \;=\; & [\{n_1, n_2\} : (d \,\square\, \widehat{n_1}); [[\{n_3, ..., n_{11}\} : (d \,\square\, \widehat{n_3} \,\square\, \widehat{n_2}) \\
& ; ((\{a, a\} \,\square\, \widehat{n_{11}} \,\square\, \widehat{n_{10}} \,\square\, \{\widehat{n_8}\} \,\square\, \{b, b, n_6, n_7\} \,\square\, \widehat{n_5} \,\square\, \{a, b, n_4\}) \\
& \parallel (\{\widehat{a}, b\} \,\square\, \widehat{n_9} \,\square\, \widehat{n_7} \,\square\, \widehat{n_6} \,\square\, \widehat{n_4})
\end{aligned}
$$

[1] X could equally have been chosen to be any of $\{t_{32}\}$, $\{t_{33}\}$ or $\{t_{34}\}$

$$\| (\{\widehat{a}, c\} \square \{c, c, n_{10}, n_{11}\} \square \{b, c, n_8, n_9\} \square \{a, c, n_5\}))$$
$$; (\widehat{d} \square n_3 \square n_2 \square n_1)] * a * [\{n_{12}\} : [(c \square \widehat{n_{12}}) * b * (\widehat{c} \square n_{12})]]]]$$

In each case, the set of transitions returned by the calls to SCOPE, (4.4), (4.5), and (4.6) is empty. Hence the recursive calls to VISIT have no further effect on the expression tree because all of the transitions in T_{sc} have already been represented by scoping operations.

During the traversal of the pruned expression tree made by VISIT, some transitions in T_{sc} are effectively ignored. The transitions that are ignored are those that were created by the action of taking two copies of a net in the semantics for the iteration operator. Hence, for example the transitions in Table 4.5 that are discarded are:

$$\{t_{31}, t_{36}, ..., t_{39}, t_{47}, ..., t_{51}\}$$

For each discarded transition t, the set $T_b(t)$ consists entirely of transitions for which the corresponding node in the expression tree was removed by the call to PRUNE. For example, $T_b(t_{31}) = \{t_3, t_{11}\}$, and it can be seen from Figures 4.18 and 4.19 that the nodes corresponding to t_3 and t_{11} were pruned. The construction by SCOPE used to represent the transition t_{28} also implicitly represents the transition t_{31} by virtue of the copy of a subnet made in constructing an implementation of an iteration expression. By looking at Figure 4.18 it can be seen that t_2 and t_{10}, which are the base transitions of t_{28}, correspond respectively to t_3 and t_{11}, the base transitions of t_{31}.

4.5 Verification of the synthesis algorithm

The proof that the synthesis algorithm presented in Sections 4.3 and 4.4 is split into two parts. The first part of the proof shows that removing the set of transitions, T_{sc}, from the input net leaves a net which is the implementation of a box expression from the basic syntax. This result is fundamental to the reuse of the basic synthesis algorithm described in Chapter 3. The second part

of the proof shows that the manipulations to the synthesised expression, made by the SCOPING synthesis rule, results in an expression whose implementation is isomorphic to the original input net. The essence of the proof is in showing that the additions made to the synthesised expression by SCOPING correspond to the creation of a new set of transitions that exactly matches T_{sc}, the set of transitions removed at the start of the synthesis process.

4.5.1 Part 1 – Removing transitions

Firstly, it is shown that every transition arising from a synchronisation operation satisfies the conditions for inclusion in T_{sc}. This means that for any implementation Σ, of a box expression, E, the transitions in $T_{sc}(\Sigma)$ can be classified into two disjoint sets, $T_{sy}(\Sigma)$, the set of transitions arising from synchronisation, and $T_{at}(\Sigma)$, the set of transitions arising from atomic actions. This classification, of course, depends on the form of the expression, E, from which Σ is derived. For example, the implementations Σ_1 of $E_1 = (a \parallel \widehat{a}) \,\Box\, \emptyset$ and Σ_2 of $E_2 = (a \parallel \widehat{a})$ **sy** a are such that $\Sigma_1 =_{iso} \Sigma_2$. However, $T_{sc}(\Sigma_1)$ consists of a single transition classified as belonging to $T_{at}(\Sigma_1)$, while $T_{at}(\Sigma_2) = \emptyset$, and the transition in $T_{sc}(\Sigma_2)$ belongs to $T_{sy}(\Sigma_2)$.

Proposition 23 *Let Σ be an implementation of a box expression, E, from the syntax in Table 4.1. Every transition that is added to Σ by the operation Σ* **sy** *a satisfies the conditions for inclusion in $T_{sc}(\Sigma$* **sy** *$a)$.*

Proof: Follows directly from the definition of T_{sc}, the iterative semantics for synchronisation, and proposition 21. □

The following propositions serve to simplify the problem by showing that for any net Σ, which is the implementation of a box expression from Table 4.1, the net $\Sigma \ominus T_{sc}(\Sigma)$ is isomorphic to $\Sigma' \ominus T_{at}(\Sigma)$, where $\Sigma' = \Sigma \ominus T_{sy}(\Sigma)$ is the implementation of a basic syntax expression. This means that the remainder of the proof in this section does not need to consider the synchronisation operator.

Some properties are shown for the net operations $\sqcup$, $\oplus$, and $\ominus$ used to give semantics to box expressions from the syntax in Table 4.1.

Proposition 24 *For disjoint nets Σ_1, Σ_2, a set of new transitions, T, such that each $t \in T$ is a multiset of transitions from Σ_1, a set of new places S, and a subset of the places of Σ_1, S', the following hold:*

$$\begin{aligned}(\Sigma_1 \oplus (T, l)) \sqcup \Sigma_2 &= (\Sigma_1 \sqcup \Sigma_2) \oplus (T, l) \\ (\Sigma_1 \oplus (T, l)) \oplus (S, l') &= (\Sigma_1 \oplus (S, l')) \oplus (T, l) \\ (\Sigma_1 \oplus (T, l)) \ominus S' &= (\Sigma_1 \ominus S') \oplus (T, l)\end{aligned}$$

Proof: Each new transition $t \in T$ is a multiset of transitions $\{t_1, ..., t_k\}$ from Σ_1, such that $t \bowtie \{t_1, ..., t_k\}$ in $\Sigma_1 \oplus (T, l)$. By the definition of the place addition operator, each $s \in S$ has the form $\{s_1, s_2\}$, where s_1 and s_2 are existing places. Hence, by their definition, the net operations $\sqcup$, $\oplus(S, l')$ and $\ominus S$ preserve the property $t \bowtie \{t_1, ..., t_k\}$. Therefore the addition of transitions commutes with net union, the addition of new places, and the removal of places. □

Proposition 25 *For any net Σ, and sets of transitions, T_1 and T_2 the following hold:*

$$\begin{aligned}\Sigma \ominus \emptyset &= \Sigma \\ \Sigma \oplus (\emptyset, l) &= \Sigma \\ \Sigma \ominus T_1 \ominus T_2 &= \Sigma \ominus (T_1 \cup T_2) \\ \Sigma \oplus (T_1, l) \ominus (T_1) &= \Sigma\end{aligned}$$

Proof: Follows directly from the definition of the $\ominus$ and $\oplus$ operators. □

Let E be any expression over the syntax in Table 4.1, and Σ be an implementation of E. The next proposition shows that, if all of the transitions that were added by the synchronisation operator during the construction of Σ are removed from Σ, then the remaining net is an implementation of E

with all of the instances of the **sy** operator removed. Given the semantics of synchronisation, this result is intuitively obvious.

Proposition 26 *Let Σ be an implementation of a box expression, E, from the syntax in Table 4.1, and E' be the expression obtained by removing every instance of* **sy** *from E. Then $\Sigma \ominus T_{sy}(\Sigma)$ is an implementation of E'.*

Proof: By structural induction over the box expression syntax. In the following let Σ be an implementation of E, and Σ' be an implementation of E', the expression obtained by removing all instances of **sy** from E. The induction hypothesis is that Σ is isomorphic to $\Sigma' \oplus (T_{sy}(\Sigma), l)$, for some labelling function l. It immediately follows that $\Sigma \ominus T_{sy}(\Sigma) =_{iso} \Sigma'$, by Proposition 25.

Base case: $E = \alpha$. By definition, $E' = E$, and any implementation of E (and therefore E'), Σ, is such that $T_{sy}(\Sigma) = \emptyset$. Hence, by Proposition 25, $\Sigma = \Sigma \oplus (T_{sy}(\Sigma), l)$.

Induction step: In the following let Σ_i and Σ_i' for $1 \leq i \leq 3$ be disjoint implementations of E_i, and E_i' respectively, where E_i' is obtained from E by removing all instances of the synchronisation operator.

- $E = E_1$ **sy** a: $\Sigma_1 =_{iso} \Sigma_1' \oplus (T_{sy}(\Sigma_1), l_1)$ follows from the induction hypothesis. By the semantics for synchronisation, $\Sigma =_{iso} \Sigma_1 \oplus (T_s, l_s)$, where T_s is the set of new transitions created by the synchronisation operation. Hence, $\Sigma =_{iso} \Sigma_1' \oplus (T_{sy}(\Sigma_1), l_1) \oplus (T_s, l_s)$. By the definition of T_{sy}, $T_{sy}(\Sigma) = T_{sy}(\Sigma_1) \cup T_s$. Therefore, by definition of the $\oplus$ operator, $\Sigma =_{iso} \Sigma_1' \oplus (T_{sy}(\Sigma), l)$ for $l = l_1 \cup l_s$.
- $E = E_1 \parallel E_2$, $E = E_1 \,[]\, E_2$, $E = E_1; E_2$: By the induction hypothesis, $\Sigma_i =_{iso} \Sigma_i' \oplus (T_{sy}(\Sigma_i), l_i)$ for $1 \leq i \leq 2$. By the semantics of parallel, choice and sequence, and Proposition 24, $\Sigma =_{iso} \Sigma' \oplus (T_{sy}(\Sigma_1), l_1) \oplus (T_{sy}(\Sigma_2), l_2)$. By the definition of T_{sy}, $T_{sy}(\Sigma) = T_{sy}(\Sigma_1) \cup T_{sy}(\Sigma_2)$. Hence, $\Sigma =_{iso} \Sigma' \oplus (T_{sy}(\Sigma), l)$, where $l = l_1 \cup l_2$.

- $E = [E_1 * E_2 * E_3]$: Follows the proof for the parallel, choice and sequence operators. It is worth noting, however, that there are two copies of each Σ_i used in the construction of Σ. Therefore, in Σ, there are two sets of transitions corresponding to $T_{sy}(\Sigma_i)$ for each $1 \leq i \leq 3$. By definition, $T_{sy}(\Sigma)$ is the union of all the $T_{sy}(\Sigma_i)$ used in the construction of Σ. Hence, there is no problem introduced by the use of two copies of each subnet in the semantics for iteration.

□

Proposition 26 shows that for any net Σ, the implementation of an expression over the syntax in Table 4.1, then $\Sigma' =_{iso} \Sigma \ominus T_{sy}(\Sigma)$ is the implementation of a basic syntax expression. The aim is to show that $\Sigma \ominus T_{sc}(\Sigma)$ is also the implementation of a basic syntax expression. A crucial observation, is that $T_{at}(\Sigma) = T_{at}(\Sigma \ominus T_{sy}(\Sigma))$. Hence, all that needs to be shown is that for any net, Σ, the implementation of a basic syntax expression, then $\Sigma \ominus T_{at}(\Sigma)$ is also the implementation of a basic syntax expression.

The following propositions characterise the form of (sub)expressions that give rise to transitions that satisfy the conditions for inclusion in T_{sc} (*i.e.* those transitions of T_{sc} that are classified as belonging to T_{at}). Firstly, it is shown that every transition that is connected to every entry and exit place in a net derived from a basic syntax box expression arises from choice composition (unless the net is an implementation of an atomic action).

Proposition 27 *Let $\Sigma = (S, T, W, \lambda)$ be an implementation of a basic syntax expression, E. If $\exists t \in T$ such that ${}^\bullet t = {}^\bullet\Sigma$ and $t^\bullet = \Sigma^\bullet$ then Σ is an implementation of either x or $E' \,[]\, x$, where $\lambda(x) = \lambda(t)$ and $\phi(x) = \{t\}$.*

Proof: By structural induction over the box expression syntax.

Base case: By definition, any implementation, Σ, of an atomic action consists of a single transition, t with ${}^\bullet t = {}^\bullet\Sigma$ and $t^\bullet = \Sigma^\bullet$.

Induction step: In the following let Σ be an implementation of E, and Σ_i for $1 \leq i \leq 3$ be disjoint implementations of E_i.

- $E = E_1 \parallel E_2$: Each connected component of Σ contains at least one entry place and one exit places. Suppose there is a transition t such that ${}^\bullet t = {}^\bullet\Sigma$, then Σ is connected. However, by the semantics of parallel composition, Σ consists of at least two disjoint components (corresponding to Σ_1 and Σ_2). Therefore there is no such transition t.

- $E = E_1 \,[]\, E_2$: By the definition of $\otimes$ and the semantics of choice composition, Σ contains a transition t such that ${}^\bullet t = {}^\bullet\Sigma$ and $t^\bullet = \Sigma^\bullet$ if and only if Σ_1 or Σ_2 contains a transition connected to every entry and exit place. Hence, by the induction hypothesis, either E_1 or E_2 has the form x or $E' \,[]\, x$, where $\phi(x) = \{t\}$. If E_1 (E_2) is x, then by the commutativity of choice composition E can be rewritten as $E_2 \,[]\, x$ ($E = E_1 \,[]\, x$). If E_1 (E_2) has the form $E' \,[]\, x$, then by the associativity and commutativity of choice composition, E can be rearranged into $E'' \,[]\, x$ where $E'' = E' \,[]\, E_2$ ($E'' = E_1 \,[]\, E'$), which is the required form.

- $E = E_1; E_2$, $E = [E_1 * E_2 * E_3]$: By Proposition 5 of Chapter 3, Σ is internally connected. Suppose there exists transition t such that ${}^\bullet t = {}^\bullet\Sigma$ and $t^\bullet = \Sigma^\bullet$. If the entry and exit places of Σ were removed, then t would become an isolated transition. Therefore, to be internally connected, Σ must consist of entry and exit places and a single transition t. By the semantics of sequential composition and iteration, Σ contains more than one transition. Hence, a contradiction has been obtained, and there is no such transition, t.

□

The next proposition shows that for any net obtained from a basic syntax expression, and any transition in that net, all the places in the pre-set of that

transition have the same label (e, $\emptyset$, or x). Similarly for the post-set of the transition.

Proposition 28 *Let $\Sigma = (S, T, W, \lambda)$ be an implementation of an expression from the syntax in Table 2.3. For every $t \in T$:*

$$\forall s_1, s_2 \in {}^\bullet t \quad : \quad \lambda(s_1) = \lambda(s_2)$$

$$\forall s_1, s_2 \in t^\bullet \quad : \quad \lambda(s_1) = \lambda(s_2)$$

Proof: By structural induction over the box expression syntax in Table 2.3, and using the semantics of box expressions. □

It can now be shown that every transition in the implementation of a basic syntax expression that satisfies the conditions for inclusion in T_{sc} (*i.e.* is a member of T_{at}) arises from an atomic action in a choice context.

Proposition 29 *Let $\Sigma = (S, T, W, \lambda)$ be an implementation of a basic syntax expression, E. If $\exists t \in T$ and $T' \subset T$ such that $|T'| \geq 2$ and $t \bowtie T'$ then E contains a subexpression $E' \,\square\, x$ (or $x \,\square\, E'$), where $\lambda(x) = \lambda(t)$ and $t \in \phi(x)$.*

Proof: By structural induction over the box expression syntax.

Base case: By definition, any implementation, Σ, of an atomic action consists of a single transition, t. Hence there is no set of transitions $T' \subseteq T$ such that $|T'| \geq 2$.

Induction step: In the following let Σ be an implementation of E, and Σ_i for $1 \leq i \leq 3$ be disjoint implementations of E_i. Let t be a transition in one of the subnets, Σ_i, used to construct Σ. The set of transitions which have at least one arc similar to t in Σ_i are given by $C = ({}^\bullet t)^\bullet \cap {}^\bullet(t^\bullet)$. Hence the set T' for Σ_i must be such that $T' \subseteq C$.

- $E = E_1 \parallel E_2$, $E = E_1; E_2$, $E = [E_1 * E_2 * E_3]$: By the compositional semantics of parallel, sequence, and iteration, the set of transitions which have at least one arc similar to t in Σ is preserved from the

subset Σ_i and given by C. Hence, by the induction hypothesis every transition that inherits the connectivity of a set of transitions arises from an atomic action in a choice context.

- $E = E_1 \,[]\, E_2$: We consider the case where $t \in T_1$. The argument is symmetric when $t \in T_2$. Suppose $T' \subseteq T_1$. By the compositional semantics of choice, $t \bowtie T'$ in Σ if and only if $t \bowtie T'$ in Σ_1. Therefore, by the induction hypothesis, t arises from an atomic action in a choice context in this case.

  Now suppose T' contains at least one transition from Σ_2. By the semantics of choice composition, the only point of contact between Σ_1 and Σ_2 is at the entry and exit interface. Hence every transition $t' \in T'$ which comes from Σ_2 must be connected only to entry and exit places. Since $t \bowtie T'$, and by proposition 28, t is connected only to entry and exit places.

  By the definition of $\otimes$, and the semantics of choice composition, every place in ${}^\bullet t' \otimes {}^\bullet\Sigma_1$ has an arc to t'. Hence, by $t \bowtie T'$, t must be connected to every entry place of Σ_1 (*i.e.* ${}^\bullet t = {}^\bullet\Sigma_1$). A similar argument shows $t^\bullet = \Sigma_1{}^\bullet$. Therefore, by proposition 27, Σ_1 is an implementation of x or $E_1' \,[]\, x$, where $\phi(x) = \{t\}$. Hence, E has the form $x \,[]\, E_2$, or $(E_1' \,[]\, x) \,[]\, E_2$.

□

Proposition 29 does not prove the existence of a transition t that has the same connectivity of a set of transitions, T'. Instead, it shows that if t does exist, then it must arise from an atomic action in a choice context. An example which demonstrates that such transitions do exist is provided by Figure 4.7 in Section 4.2 where $t_5 \bowtie \{t_1, t_4\}$. The general form of basic syntax expressions that give rise to transitions that have the same connectivity as a set of transitions is:

$$E = (E_1 \parallel E_2 \parallel \ldots \parallel E_n) \,[]\, x$$

where each E_i has the form x_i or $E_i' \,[]\, x_i$. For any implementation of E, the set of transitions $T' = \{\phi(x_1), ..., \phi(x_n)\}$, is such that $\phi(x) \bowtie T'$. Proposition 26 shows that removing the set of transitions classified as belonging to T_{sy} leaves the implementation of a basic syntax expression. The following proposition is an analogue of Proposition 26, applied to those transitions classified as belonging to T_{at}.

Proposition 30 *Let Σ be an implementation of a box expression, E, from the basic syntax. Then $\Sigma \ominus T_{at}(\Sigma)$ is an implementation of a basic syntax box expression.*

Proof: By Proposition 29, and the commutativity of the choice operator, every transition in $t \in T_{at}(\Sigma)$ arises as the result of a (sub)expression of the form $E \,[]\, x$, where $\lambda(x) = \lambda(t)$ and $t \in \phi(x)$. Let Σ be an implementation of E. By the semantics of choice composition, $\Sigma \oplus (T_b(t), \lambda(t))$ is an implementation of $E \,[]\, x$. Hence, it has been shown that every transition belonging to T_{at} can be represented semantically in the same form as transitions arising from synchronisation. The remainder of the proof follows that of Proposition 26, and is a consequence of the commutativity of the $\oplus$ operator with the other net operators used to implement the semantics for basic syntax expressions. Therefore, $\Sigma \ominus T_{at}(\Sigma)$ is an implementation of a basic syntax box expression. □

Propositions 26 and 30 are combined in Theorem 3 to give the result that removing the set of transitions $T_{sc}(\Sigma)$ from the input net, Σ, leaves a net that is a suitable input to the synthesis algorithm of Chapter 3.

Theorem 3 *Let $\Sigma = (S, T, W, \lambda)$ be the implementation of a box expression, E from the syntax in Table 4.1. The net $\Sigma \ominus T_{sc}(\Sigma)$ is the implementation of an expression from the syntax in Table 2.3.*

Proof: By proposition 26, $\Sigma \ominus T_{sc}(\Sigma) =_{iso} \Sigma' \ominus T_{at}(\Sigma)$, where Σ' is the implementation of a basic syntax box expression, and $T_{at}(\Sigma)$ the set of

transitions that satisfy the conditions for inclusion in $T_{sc}(\Sigma)$, and have arisen from atomic actions in E. Since T_{at} and T_{sy} are disjoint sets of actions, $T_{at}(\Sigma) = T_{at}(\Sigma')$, and $\Sigma \ominus T_{sc}(\Sigma) =_{iso} \Sigma' \ominus T_{at}(\Sigma')$. By Proposition 30, $\Sigma' \ominus T_{at}(\Sigma')$ is the implementation of a basic syntax expression. □

4.5.2 Part 2 - Adding transitions back again (Soundness)

Proposition 31 shows that the expression tree constructed by the start of line 5 of BOX EXPRESSION SYNTHESIS corresponds to the net $\Sigma \ominus T_{sc}$. This section shows that the call to SCOPING in line 5 modifies the expression tree in such a way that exactly those transitions in T_{sc} are represented, and the modified expression tree corresponds to the input net, Σ.

Consider the code for BOX EXPRESSION SYNTHESIS in Section 4.3.1. A corollary of Theorem 3, is that the net used to initialise *N.net* in line 2, can be synthesised to an expression using the basic syntax synthesis algorithm described in Chapter 3. Firstly, it is shown that lines 3 and 4 of BOX EXPRESSION SYNTHESIS result in an expression tree the same as the one that would be produced using the synthesis algorithm of Chapter 3 (apart from, of course, the addition of the *Partition* field in each node in the tree).

Proposition 31 *For any net, Σ obtained from an expression over the syntax in Table 4.1, the expression tree obtained by the end of line 4 of* BOX EXPRESSION SYNTHESIS(Σ) *in Section 4.3.1 is the same as the expression tree obtained from the* CANONICAL BOX EXPRESSION SYNTHESIS *algorithm of Chapter 3 on input $\Sigma \ominus T_{sc}(\Sigma)$, provided the same total ordering of box expressions is used. Two trees are considered "the same" when they are identical if the* Partition *fields of nodes are ignored.*

Proof: Follows from the similarity of SYNTHESISE in Section 4.3.1 and ORDERED SYNTHESISE in Chapter 3, and the fact that PRUNE removes

exactly those additional subtrees generated by the modified iteration synthesis rule.

The modified iteration synthesis rule generates an ordered set of six subnets (rather than an ordered set of three subnets). For the purposes of this proof, these subnets will be named $\Sigma_1, ..., \Sigma_6$. The first three subnets, Σ_1 to Σ_3 are obtained in exactly the same way as in the original iteration synthesis rule in Chapter 3. The expression subtrees synthesised from second set of three subnets, Σ_4 to Σ_6 are those that are discarded by PRUNE. □

It is the construction of the *Partition* fields that require the two steps (SYNTHESISE and PRUNE) to generate an expression tree. The *Partition* field of a node N encodes a relationship between atomic actions in the expression represented by the (sub)tree with root N, and the transitions in the net stored in the *Net* field of N. It has already been seen that in constructing the net corresponding to a particular expression, several transitions may arise from each atomic action in the expression. The *Partition* field encodes a mapping representing one way in which the transitions in the net may arise from the atomic actions in the synthesised expression.

The following proposition demonstrates that the bottom-up construction of the *Partition* fields by SYNTHESISE and PARTITION provides a valid mapping from atomic actions in the synthesised expression to transitions in the net being synthesised.

Proposition 32 *The partition field of each node in the synthesised expression tree created by* SYNTHESISE *represents a valid mapping between atomic actions and transitions in the net field of that node. A valid mapping is one where there exists a construction of the net from the expression such that for each atomic action, α in the expression, the set of transitions corresponding to α (given by the mapping) arise from α.*

Proof: By structural induction over the synthesised expression.

Base case: The modified atomic action synthesis rule of Section 4.3.3 sets the partition field of the node to $[\{t_1\}]$ where t_1 is the name of the transition in the net being synthesised. Clearly, $\{\alpha\} \rightarrow \{t_1\}$ is a valid mapping for $E = \alpha$.

The mapping represented by the partition field is from atomic actions to sets of transitions. The partition field itself is a sequence of sets of transitions. The correspondence between an atomic action and a set of transitions in the partition field can be found by traversing the expression tree in a depth-first fashion. The set of transitions corresponding to first atomic action visited is given by the first entry in the sequence, and so on. Once consequence of this representation for the mapping is that it is no longer possible to manipulate the expression tree so easily. For example, to change the order of the children of a node it is necessary to update the partition field of that node, and of all the ancestors of the node. This is why a canonical ordering is imposed in line 11 of SYNTHESISE before the partition fields are constructed for that node, and its ancestors.

Induction step: In the following let $\phi_1, ..., \phi_n$ be the partition fields of the nodes $N_1, ..., N_n$, respectively, where $N_1, ..., N_n$ are the children of the node, N, in order. By the induction hypothesis, ϕ_i is a valid mapping for N_i, for $1 \leq i \leq n$. Consider the type of connective that is represented by node N.

- *N.type* = *parallel*, *N.type* = *choice*, *N.type* = *sequence*: Performing a depth-first search starting a N visits the nodes of N_1 in depth-first order, followed by those of $N_2, ..., N_n$. Hence, a valid mapping is given by concatenating the sequences $\phi_1, ..., \phi_n$ in order. This is the partition field created by line 7 of PARTITION.

- *N.type* = *iteration*: For the expression $E = [E_1 * E_2 * E_3]$, two copies of each of the nets corresponding to, E_1, E_2, E_3 are used the the construction of an implementation of E. The modified synthesis

rule for iteration, described in Section 4.3.3, synthesises expression trees $N_1, ..., N_6$. N_1 and N_4 both represent the subexpression E_1, derived from the two copies of the net corresponding to E_1. Since canonical form expressions are synthesised, the structure of the subtrees with roots N_1 and N_4 will be identical. Hence the order in which atomic actions are visited in a depth-first search of N_1 and N_4 will be the same. Therefore, the mapping given by $\phi_1 \cup \phi_4$, where $\cup$ is the list union operation defined in in Section 4.3.4 is a valid mapping for E_1 in $[E_1 * E_2 * E_3]$. A similar argument can be applied to the pairs of subtrees N_2, N_5 and N_3, N_6.

□

The definitions of synchronisation in Chapter 1, and Section 4.2.1 have been given in terms of the synchronisation of transitions in nets. The following proposition formalises the relationship between the notion of synchronisation of sets of actions, and the corresponding synchronisation of transitions, using the semantics for synchronisation given in Section 1.3.5 in Chapter 1.

Proposition 33 *Let $E = E_1$* **sy** *a be a synchronisation expression, and $\Sigma = (S, T, W, \lambda)$ be an implementation of E_1, and τ be a finite multiset of the set of transitions, T. The multiset of actions in E_1, corresponding to τ is given by:*

$$\tau^A = \{\phi^{-1}(t) \mid t \in \tau\}$$

where ϕ is the mapping from actions in E_1 to transitions in Σ, as defined in Section 2.11. If τ is a valid synchronisation, then every synchronisation in:

$$T_\tau = \phi(\alpha_1) \odot \phi(\alpha_2) \odot ... \odot \phi(\alpha_n)$$

where $\tau^A = \{\alpha_1, ..., \alpha_n\}$, is also valid. Furthermore, any synchronisation operation applied to E_1 where τ is a valid synchronisation, will necessarily create all the synchronisations in T_τ.

Proof: The validity of a synchronisation, τ, depends only on the labels of the transitions in τ. By the definition of ϕ, for an atomic action, α, every transition in $\phi(\alpha)$ has the same label. Therefore, each synchronisation in T_τ is valid if and only if τ is valid. Suppose τ is a valid synchronisation, then every atomic action in τ^A must be in the scope of the synchronisation operator. The set of synchronisations, T_τ consist entirely of transitions derived from actions in τ^A. Therefore, every synchronisation in T_τ is valid. □

Corollary 3 *The set of transitions arising from synchronisation can be partitioned into groups according to the corresponding synchronisation of multisets of actions in the expression from which they were derived.*

The following proposition is central to the approach used to synthesise synchronisation. It shows that the scoping operator can be used to represent transitions arising from the synchronisation operator. In fact, the stronger result that the set of transitions, T_{sc} can be represented using the scoping operator is shown.

Proposition 34 *Let E be an expression from the syntax in Table 4.1, and Σ be an implementation of E. There exists an expression, E', from the syntax in Table 4.3, such that the implementation of E' is isomorphic to Σ, and every transition in $T_{sc}(\Sigma)$ results from an application of the scoping operator.*

Proof: By induction over the structure of E, it is shown that every syntactic structure giving rise to transitions belonging to $T_{sc}(\Sigma)$, has an equivalent form which uses the scoping operator.

Base case: The implementation of α does not contain any transitions that satisfy the conditions for inclusion in T_{sc}.

Induction step:

- $E = E_1 \,\square\, \alpha$: By Proposition 29, the transition, t, arising from α may satisfy the conditions for inclusion in T_{sc}. If so, there is a

set of atomic actions in E_1 which give rise to the transitions $T_b(t)$. By definition of T_b, no transition in the set $T_b(t)$ is a member of T_{sc}. Hence, there is an atomic action in E_1 corresponding to each transition in $T_b(t)$. Therefore suppose $T_b(t) = \{t_1, ..., t_n\}$, and the actions $\alpha_1, ..., \alpha_n$ in E_1 correspond to $t_1, ..., t_n$. E_1' is constructed from E_1 by introducing $n-1$ new basic actions, $n_1, ..., n_{n-1}$, which are not used elsewhere, and replacing each action α_i in E_1, for $1 \leq i \leq n-1$ by $(\alpha_i \,\square\, \{\widehat{n_i}\})$. Finally, the action, α_n is replaced by $(\alpha_n \,\square\, \{n_1, ..., n_{n-1}\} + \lambda(t))$. The modified form of E is given by $E' = [\{n_1, ..., n_{n-1}\} : E_1']$. By the semantics of choice, the implementation of E_1' is isomorphic to the implementation of E_1 with transitions duplicating $\alpha_1, .., \alpha_n$ added. By the definition of scoping, the implementation of E' is constructed from an implementation of E_1' by synchronising then restricting on the set of actions, $n_1, ..., n_{n-1}$. By the semantics of synchronisation, only one transition which does not contain any of $n_1, ..., n_{n-1}$ is created by the synchronisation operation. This transition, t', is obtained by synchronising the n new transitions added to E_1, and has the label $\lambda(t)$. Therefore, the effect of the scoping operation on E_1' is to remove the n transitions that were added, and create a single new transition t' such that $t' \bowtie T_b(t)$. Hence, the implementation of E' is isomorphic to Σ.

- $E = E_1$ **sy** A: By Proposition 23, every transition created by the synchronisation operation satisfies the conditions for inclusion in T_{sc}. For each transition, t created by the synchronisation operation, there is a corresponding set of base transitions, $T_b(t) = \{t_1, ..., t_n\}$. Let $\alpha_1, ..., \alpha_n$ be the actions in E_1 corresponding to $t_1, ..., t_n$. By Proposition 33 and Corollary 3, there will be a set of transitions in T_{sc} corresponding to each of the synchronisations derivable from $\alpha_1, ..., \alpha_n$ (*i.e.* $\phi(\alpha_1) \odot \phi(\alpha_2) \odot ... \odot \phi(\alpha_n)$). This set of transitions

can be dealt with simultaneously using the construction described above. Repeated application of this process can be used to deal with all of the transitions created by the synchronisation operation.

- E has a form different from the two cases above: By Propositions 26 and 29, E does not give rise to any transitions that satisfy the conditions for inclusion in T_{sc}.

□

The construction described in Proposition 34 inserts scoping operators at the positions in the expression where the original choice or synchronisation operations were. However, SCOPING and VISIT only consider adding scoping operators at the root and as children of iteration nodes in the expression tree. The following proposition justifies this restriction on the position of scoping operators.

Proposition 35 *The only positions that need to be considered for adding the scoping operators in* SCOPING *and* VISIT *are as the root node of the expression tree, and as children of an iteration node.*

Proof: Follows from the soundness of the following axioms:

$$
\begin{aligned}
[N_1 : [N_2 : E]] &= [N_1 \cup N_2 : E] \text{ provided } N_1 \cap N_2 = \emptyset \\
[N : E_1] \parallel E_2 &= [N : E_1 \parallel E_2] \text{ provided } \forall n \in N : \mathcal{L}(E_2) \cap \{n, \hat{n}\} = \emptyset \\
[N : E_1] \,\Box\, E_2 &= [N : E_1 \,\Box\, E_2] \text{ provided } \forall n \in N : \mathcal{L}(E_2) \cap \{n, \hat{n}\} = \emptyset \\
E_1; [N : E_2] &= [N : E_1; E_2] \text{ provided } \forall n \in N : \mathcal{L}(E_1) \cap \{n, \hat{n}\} = \emptyset \\
[N : E_1]; E_2 &= [N : E_1; E_2] \text{ provided } \forall n \in N : \mathcal{L}(E_2) \cap \{n, \hat{n}\} = \emptyset
\end{aligned}
$$

Each set of new basic actions used to create a transition via scoping is unique, and those basic actions are not used elsewhere. Therefore, there is freedom in positioning of the scoping operator, provided:

- The scoping operator encloses all of the transitions that contain the basic actions on which scoping is performed.

- The scoping operator does not move across an iteration operation because the mapping between atomic actions and transitions changes across iteration, and this affects the set of transitions created by the scoping operator.

□

It remains to show that the scoping synthesis rule partitions the set of transitions, T_{sc}, so that each partition of T_{sc} can be represented using the scoping operator.

Theorem 4 *The call to* SCOPING *in line 5 of* BOX EXPRESSION SYNTHESIS *modifies the synthesised expression tree so that it represents the set of transitions, T_{sc} that was removed from the input net.*

Proof: By induction on the depth of the candidate location for the insertion of a scoping operator.

Base case: Transitions in T_{sc} dealt with by a scoping operator inserted at the root node of the expression tree. By Propositions 32 and 33, for any candidate transition, $t \in T_{sc}$, the set of actions in the synthesised expression that should be synchronised to generate t is given by $\phi^A = \{\phi^{-1}(t') \mid t' \in T_b(t)\}$. By Proposition 33 and the definitions of synchronisation and $\odot$, the set of transitions that would be created if $\phi^A = \{\alpha_1, ..., \alpha_n\}$ were synchronised is given by $T_t = \phi(\alpha_1) \odot ... \odot \phi\alpha_n$, where each transition in T_t has the same label as t. If every transition in T_t appears in T_{sc}, then by adding a scoping operation to create t the group of transitions $T_t \subseteq T_{sc}$ will be dealt with. If not every transitions appears, it follows by Proposition 34 that the set of transitions $T_{sc} \cap T_t$ must be dealt with by a scoping operator inserted at a lower level in the expression tree. Each iteration of the **while** loop of SCOPE chooses a candidate transition from those remaining to be dealt with. If the group of transitions that would be generated appears in Tr, then the candidate

transition is added to X, and the group of transitions removed from Tr. Once all of the groups in Tr have been checked, those that can be dealt with are represented by modifications to the expression tree, as described in Proposition 34. The remaining transitions are returned to the calling procedure to be dealt with at a lower level in the expression tree.

Induction step: By the induction hypothesis, all transitions that can be dealt with at a higher level have been removed from the set of transitions remaining to be represented, Tr. It follows that when considering inserting a scoping operator above a node N in the expression tree, those transitions $t \in Tr$, such that the atomic actions corresponding to each $t' \in T_b(t)$ appear as descendents of n, can be considered as candidates to be represented at this location. By Proposition 35 the locations for inserting scoping operators considered by SCOPING and VISIT are sufficient to deal with every transition in T_{sc}. The argument given above for the base case can be used again to show that the groups of transitions that can be represented at each candidate location are identified, and the expression tree is modified to represent those groups of transitions. □

Corollary 4 *Given any finite net, Σ, which is derived from an expression over the syntax in Table 4.1, a call to* BOX EXPRESSION SYNTHESIS*(Σ) synthesises an expression from the syntax in Table 4.3 whose implementation is isomorphic to Σ.*

4.6 Related problems

The time complexity of the algorithm for BOX EXPRESSION SYNTHESIS presented in this chapter is shown to be polynomial in Section 4.6.1. Section 4.6.2 analyses the areas of non-determinism in the algorithm and presents some evidence that modifying the algorithm to efficiently produce canonical form

expressions may be difficult. A sound axiom system is introduced in Section 4.6.4, and the analysis of non-determinism in Section 4.6.2, together with the results from Section 4.5 are used to show that the axiom system is complete.

4.6.1 Time complexity

The analysis of the time complexity in this section is based on the size of the input net $\Sigma = (S, T, W, \lambda)$. Recall that infinite synchronisations are not considered, and hence Σ is finite in size. For simplicity, it is assumed that the size of each transition label is bounded by some constant. Let $n = |S| + |T|$ be the number of nodes in Σ. There is at most one arc between any pair of nodes (although this arc may have a weight), and Σ is bipartite, with bipartition S, T. Therefore the number of arcs in Σ is at most $|S| \cdot |T| < n^2$. Hence, it is sufficient to consider the time complexity of the algorithm in terms of the number of nodes, n.

The synthesis algorithm of this chapter is largely based on the basic synthesis algorithm of Chapter 3. In extending the analysis of the time complexity of the basic synthesis synthesis to the algorithm of this chapter, the following areas are considered:

- The computation of the set of transitions, T_{sc}, removed from the input net in Line 2 of BOX EXPRESSION SYNTHESIS in Section 4.3.1.

- The synthesis of an underlying expression, and the computation of the partition information associated with the nodes in the expression tree representing the underlying expression.

- The complexity of SCOPING (*i.e.* SCOPE, VISIT and INSERT SCOPING) which modify the expression tree to represent the set of transitions T_{sc} that were removed from the input net.

An investigation into the time complexity of each of the above areas is presented, followed by an overall analysis of the time complexity for the synthesis algorithm.

Time complexity of computing T_{sc}

The method for computing T_{sc} described and analysed in this section is chosen so that the base transitions $T_b(t)$ for $t \in T$ can also be found, with relatively little extra work. The base transitions are used later in the SCOPE and INSERT SCOPING procedures. The most efficient approach is to compute T_b once at the start of the algorithm and store the results for later use, rather than recompute T_b each time it is needed.

In the analysis below, it is assumed that there is a total order, $<_l$ over the transitions in the input net, such as that defined in Section 2.5.6, and the comparison $t_1 <_l t_2$ between transitions t_1 and t_2 takes $O(1)$ time[2]. The following describes the steps that may be used to compute the set of transitions, T_{sc}, and the function T_b.

The transitions are grouped into equivalence classes, according to the relation $\sim_{dpl}$, defined in Section 2.5.4 in Chapter 2. Testing whether $t_1 \sim_{dpl} t_2$ for transitions t_1 and t_2 takes $O(n)$ time[3]. To compute the equivalence classes of $\sim_{dpl}$, comparing every pair of transitions in the input net is sufficient. Hence the time complexity is $O(n^3)$. Note that it is also possible to find the canonical representative for each equivalence class, based on the ordering $<_l$, without any impact on the overall time complexity.

In computing T_{sc}, only the canonical representative from each equivalence class of $\sim_{dpl}$ need be considered, since if $t_1 \in T_{sc}$ and $t_1 \sim_{dpl} t_2$, then it immediately follows that $t_2 \in T_{sc}$. This optimisation does not have any effect

[2]This is a safe assumption provided there is some fixed upper bound on the size of transition labels.

[3]The analysis in this section assumes an adjacency matrix representation for the net. It is likely that in practice that an adjacency list representation would be much more efficient.

on the theoretical time complexity of the algorithm as there will be $O(n)$ equivalence classes of $\sim_{dpl}$.

To check whether a transition, t belongs to T_{sc}, the brute force approach of taking every pair of transitions t_1, t_2 and testing whether $t \bowtie \{t_1, t_2\}$ may be used. Checking if such a pair of transitions exists takes $O(n^3)$ time. Therefore, computing the set of transitions, T_{sc} has time complexity $O(n^4)$. For each transition t, found to belong to T_{sc}, the canonical representatives of the equivalence classes of $\sim_{dpl}$ to which t_1 and t_2 belong, may be stored without any additional impact on the time complexity.

T_b may be calculated using the recursive definition given in Section 2.5.5, and the information computed in the previous steps. As above, T_b only needs to be found for the canonical representative of each equivalence class of $\sim_{dpl}$.

It is important to place a bound on the size of $T_b(t)$ for each $t \in T_{sc}$. From the definition of T_b, it appears that T_b may not be computable efficiently, and that an exponential number of transitions may be introduced to represent a transition $t \in T_{sc}$, using the scoping operator. However, it can be shown that for any $t \in T$ $|T_b(t)| < n$:

Proposition 36 *For any $t \in T$, $|T_b(t)| \leq |T|$.*

Proof: The input net $\Sigma = (S, T, W, \lambda)$ is derived from some expression, E, from the Box Expression syntax in Table 4.1. By induction over the structure of E, it is shown that every syntactic structure gives rise to transitions t that satisfy the property $|T_b(t)| \leq |T|$.

Base case: $E = \alpha$. By the semantics of atomic actions, $|T| = 1 (= \{t\})$, and by the definition of T_b, and Propositions 26 and 29, $|T_b(t)| = 1$. Hence, $|T_b(t)| \leq |T|$.

Induction step: By the induction hypothesis, every transition t_i in the implementation $\Sigma_i = (S_i, T_i, W_i, \lambda_i$ of E_i (for $i = 1, 2, 3$) is such that $|T_b(t_i)| \leq |T_i|$. It is shown that for every transition t in the implementation, $\Sigma = (S, T, W, \lambda)$ of E, $|T_b(t)| \leq |T|$.

- $E = E_1 \parallel E_2$, $E = E_1; E_2$, $E = [E_1 * E_2 * E_3]$: By Propositions 26 and 29, every transition in T keeps the same set of base transitions as in Σ_i ($i = 1, 2$ or 3). That is, suppose $t \in T$ (in Σ) arises from $t_i \in T_i$ (in Σ_i), then $T_b(t)$ in Σ is the same set as $T_b(t_i)$ in Σ_i. Therefore, it immediately follows that for all $t \in T$, $|T_b(t)| \leq |T|$.

- $E = E_1 \,[]\, E_2$: By Proposition 29, it is possible that some $t \in T$, satisfies the conditions for inclusion in T_{sc}. By Proposition 29, every arc connected to t has weight 1 (*i.e.* $\forall x \in S \cup T, W(t,x) + W(x,t) \leq 1$). Therefore, every transition in $T_b(t)$ must be different, and so $|T_b(t)| \leq |T|$.

- $E = E_1$ **sy** a: Consider a transition $t \in \Sigma$, which arises as a result of the synchronisation operation. It follows from the iterative semantics of synchronisation presented in Section 4.2.1 that there are a pair of transitions t_1, t_2 that synchronise to produce t. Without loss of generality, suppose that there is an $a \in \lambda(t_1)$, and a $\widehat{a} \in \lambda(t_2)$ which are used to synchronise t_1 and t_2 to obtain t. By the assumption that the input net to the synthesis algorithm is finite, $t_1 \neq t_2$. Consider a transition $t' \in T_b(t)$, and suppose $T_b(t)(t') = 1$, then the problem reduces to showing that $|T_b(t) - \{t'\}| \leq |T - \{t'\}|$. Now suppose that $T_b(t)(t') = x$. By the definition of the semantics for synchronisation in Section 1.3, it follows that there are valid synchronisations $ts_1, ..., ts_{x-1}$ such that $T_b(ts_i)$ is the same as $T_b(t)$, except that there are i copies of t' rather than x copies. Hence, the problem reduces to showing that $|T_b(t) - x \cdot \{t'\}| \leq |T - \{t', ts_1, ..., ts_{x-1}\}|$. Therefore, $|T_b(t)| \leq |T|$.

□

For the analysis of the time complexity of SCOPING, it is useful to be able to place a bound on the number of additional actions that will be added to the synthesised underlying expression to represent the set of transitions, T_{sc}.

Corollary 5 *The bound on the number of actions that will be added to the synthesised underlying expression is* $O(n^2)$.

Proof: The number of actions in T_{sc} is $O(n)$. By Proposition 36, for each transition $t \in T_{sc}$, the number of actions in $T_b(t)$ is $O(n)$. By the code for SCOPE, it can be seen that INSERT SCOPING is called at most once to deal with each transition $t \in T_{sc}$, and from INSERT SCOPING, the number of new actions that are added to represent t in the synthesised expression is $|T_b(t)|$. Therefore, the bound on the number of new actions required to deal with all the transitions is T_{sc} is $O(n^2)$. □

Time complexity of synthesising underlying expression

This section considers the time complexity of the calls to SYNTHESISE and PRUNE in Lines 3 and 4 of BOX EXPRESSION SYNTHESIS. As has already been noted the synthesis algorithm is largely based on that of Chapter 3. There are three differences in the SYNTHESISE procedure:

- The modified atomic action synthesis rule. It takes $O(1)$ time to initialise the partition field of the node. Hence, the time complexity, of the atomic action rule remains at $O(1)$, the same as in Table 3.4 in Chapter 3.

- The modified iteration synthesis rule. The synthesis rule described in Chapter 3 decomposes the iteration net into two isomorphic subnets. One of these nets is discarded, and the other is decomposed further into three components. The analysis in Chapter 3 shows that these two steps take $O(n^3)$ time. The modified iteration rule of this chapter decomposes both of the isomorphic subnets obtained from the first decomposition. Hence, the time complexity is $2.O(n^3)$, *i.e.* it remains at $O(n^3)$.

  There is a further consideration associated with the modified iteration synthesis rule. Clearly the modification results in a much greater amount of work for the synchronisation synthesis algorithm compared to the basic

synthesis algorithm of Chapter 3, since there are an extra three subnets to synthesise expressions for after an application of the iteration synthesis rule. However, the analysis of time complexity in Chapter 3 does not take into account the fact that part of the input net is discarded during an application of the iteration synthesis rule in the basic synthesis algorithm. Hence, the overall time complexity of SYNTHESISE as analysed in Chapter 3 is not impacted by the modified iteration synthesis rule.

- Computing the partition fields. A call to PARTITION is made after each application of a synthesis rule other than the atomic action rule. By definition, the size of the partition field in a node is bounded by the number of transitions in the net corresponding to the expression represented by the node. When the call to PARTITION is made after the application of the parallel, choice or sequence synthesis rules, the partition fields are appended to each other, taking $O(n)$ time (assuming a linked list representation for the partition field). After the iteration synthesis rule has been applied, the call to PARTITION performs three list union operations (each taking $O(n)$ time).

  Hence, the inclusion of the call to PARTITION in SYNTHESISE adds an additional $O(n)$ time to the application of each synthesis rule, apart from the atomic action rule. Since these synthesis rules have a time complexity of at least $O(n^2)$, there is no impact on the overall time complexity of the synthesis algorithm.

The most efficient way to implement PRUNE in line 4 of BOX EXPRESSION SYNTHESIS is to deleted the second set of three subtrees of an iteration node in the PARTITION procedure. The only purpose of synthesising expressions for both copies of the subnets of a iteration net is to compute the correct partition field for the iteration node. Once the partition field of the iteration node has been computed, the three subtrees can safely be removed from the expression tree. Marking a subtree as deleted can be done in $O(1)$ time. Hence, PRUNE

does not have any impact on the overall time complexity of the synthesis algorithm.

Time complexity of SCOPING

The call to SCOPING in line 5 of BOX EXPRESSION SYNTHESIS uses VISIT to perform a depth-first traversal of the underlying expression tree and calls SCOPE at the root node, and the three nodes immediately below each iteration node. The time complexity of the depth first traversal is $O(n^2)$ [4], and SCOPE will be called $O(n)$ times.

The size of the set of transitions passed to SCOPE is at most $|T_{sc}|$ (*i.e.* $O(n)$), and decreases as the transitions in T_{sc} are represented by additions to the expression tree. Therefore, the while loop in lines 3-10 is executed $O(n)$ times during each call to SCOPE.

It is not clear whether T_b' in line 5 of SCOPE can be computed efficiently. However, notice that T_b' is only used in the construction of T' in line 6, so the assumption is made that T_b' is not constructed explicitly, when considering the time complexity of finding T'. Instead, the multiset of sets of transitions, $\phi(t_1), ..., \phi(t_n)$ is constructed, taking $O(n^2)$ time.

Consider t', a candidate transition for inclusion in T'. By the assumption on transition label sizes, it takes $O(1)$ time to compare $\lambda(t')$ and $\lambda(t)$. By the construction used to find T_{sc}, the value of $T_b(x)$ is already available for every transition $x \in T$.

From the definition of ϕ, it follows that:

$$(x_1 \in \phi(t_1)) \wedge (x_1 \in \phi(t_2)) \Leftrightarrow t_1 = t_2$$

Therefore, to check whether $\exists A \in T_b' : T_b(t') = A$, it is sufficient to mark every transition in the sets $\phi(t_1), ..., \phi(t_n)$ that also appears in $T_b(t')$, then $\exists A \in T_b' : T_b(t') = A$ if and only if at least one transition in each $\phi(t_i)$ for

[4]This is based on Corollary 5, which shows that by the completion of the synthesis algorithm, the bound on the size of the expression tree will grow to $O(n^2)$

$1 \leq i \leq n$ is marked. The time complexity of this check is $O(n^3)$. Lines 7-10 of SCOPE require $O(n)$ time. Hence, the while loop of SCOPE has time complexity $O(n^4)$.

The size of the set of transitions passed to INSERT SCOPING is $O(n)$. For each iteration of the loop, the construction of L in lines 3-7 of INSERT SCOPING takes $O(n)$ time. The depth-first traversal of the expression tree takes $O(n^2)$ time, although at most $O(n)$ nodes will pass the test in line 9 (assuming new transitions added by ADD are marked so that they are ignored). The while loop in lines 10-12 has time complexity $O(n^2)$, with calls to both ADD and ADD SCOPING requiring $O(1)$ time. Therefore, the overall time complexity of both INSERT SCOPING, and therefore SCOPE is $O(n^4)$. Hence, the time complexity of SCOPING is $O(n^5)$.

Time complexity of BOX EXPRESSION SYNTHESIS

Table 4.11 summarises the analysis of the time complexity of BOX EXPRESSION SYNTHESIS carried out above.

Line	Time complexity
1 N=*new* node	$O(1)$
2 N.net=$\Sigma \ominus T_{sc}$	$O(n^4)$
3 SYNTHESISE(N)	$O(n^5)$
4 PRUNE(N)	not applicable
5 SCOPING(N, Σ)	$O(n^5)$
6 **return** EXPRESSION(N)	$O(n^2)$

Table 4.11: Time complexity of BOX EXPRESSION SYNTHESIS

The analysis of BOX EXPRESSION SYNTHESIS has shown that the theoretical bound on time complexity is not any greater for the synchronisation synthesis algorithm than for the basic synthesis algorithm in Chapter 3. This is perhaps a reflection that the bound on time complexity found here and in

Chapter 3 is rather loose. The important result of the analysis carried out in this section is that the time complexity is polynomial, and hence the algorithm is efficient.

4.6.2 Non-determinism

In this section, the points of non-determinism in the synthesis algorithm for synchronisation are investigated. The purpose of the investigation is to provide a basis for the production of a sound and complete axiom system. Consideration is also given to the possibility of extending the algorithm so that a canonical form expression is synthesised.

Synthesis of underlying expression

There is one important source of non-determinism in lines 1-4 of BOX EXPRESSION SYNTHESIS. The set of transitions, T_{sc}, used in line 2 is uniquely defined for any input net, Σ, and the synthesis algorithm for the underlying net produces a canonical form expression tree. However, in removing T_{sc} from the input net, Σ, some information about the structure of Σ is lost. The result of this is some non-determinism introduced into the step which synthesises the underlying expression tree.

For example, the implementation of $E = (a;\widehat{a}) \parallel ((a;\widehat{a})\ \mathbf{sy}\ a)$ is shown Figure 4.23, with the transition belonging to T_{sc} indicated by dotted lines. It can be seen that once the set of transitions, T_{sc}, has been removed from the net, the two disjoint subnets are isomorphic to each other. Hence, when synthesising the underlying expression, the ordering of the subexpressions corresponding these subnets does not affect the canonical form of the expression. The ordering does become important, however, once the SCOPING procedure is called and further actions are added to the underlying expression. This source of non-determinism means that the net in Figure 4.23 could equally be

synthesised to either of the following expressions:

$$E = [\{n_1\} : ((n_1 \,\square\, a); (\widehat{a} \,\square\, \widehat{n_1})) \parallel (a; \widehat{a})]$$

$$E = [\{n_1\} : (a; \widehat{a}) \parallel ((n_1 \,\square\, a); (\widehat{a} \,\square\, \widehat{n_1}))]$$

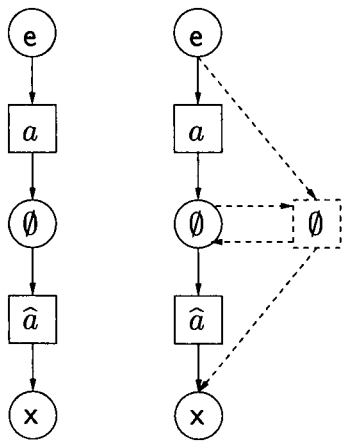


Figure 4.23: Non-determinism as a result of removing T_{sc}

Of course, the example in Figure 4.23 is very simple. When each subnet contains many transitions belonging to T_{sc}, and there are transitions in T_{sc} linking subnets together, then the problem becomes much more complex.

SCOPING procedure

There are several points of non-determinism in SCOPING. Each of these are considered in turn below. An assessment is given of whether each point of non-determinism has any effect on the output of the synthesis algorithm, and if so, how difficult it would be to remove.

The while loop in SCOPE considers transitions $t \in U$ in an arbitrary order. Lines 5 and 6 determine whether t can be represented by a scoping operator at the current location in the expression tree. The check in lines 5 and 6 is independent of choice of order in which transitions from U are considered. However, note that in line 10 of SCOPE, several transitions may be removed from U for each iteration of the while loop. Hence, the order in which transitions are considered from U, will affect the resulting set of transitions X.

Recall that each $t \in X$ is a representative of a set of transitions that all have the same base transitions. It can be seen from INSERT SCOPING that it is the base transitions of $t \in X$ that are used to determine the location for the addition of new actions to the expression tree, rather than t itself. Therefore, the order in which transitions from U are considered does not affect the form of the resulting synthesised expression.

The definition of the set of base transitions, $T_b(t)$, for each $t \in T$, is based on an ordering, $<_t$ over the transitions in the input net. To some extent, $<_t$, defined in Section 2.5.6, is based on transition names. Hence, it is conceivable that the choice of transition names may influence the form of the synthesised expression. T_b is used in two places as a result of a call to SCOPING:

- In line 5 of SCOPE, $T_b(t)$ is used in the construction of T'_b, a multiset of transitions that determines whether t may be represented by a scoping operator at the current point in the expression tree. This use of T_b is not sensitive to any change in the transition names of the input net because such a change will affect $T_b(t')$ and $T_b(u)$, used in line 6 of SCOPE, in the same way.

- T_b is used in the construction of L in line 3 of INSERT SCOPING. L determines the locations in the expression tree where new actions are inserted using ADD. Suppose that the choice of transition names in the input net affects the ordering of t_1 and t_2, and causes a new action to be inserted at the action corresponding to t_2 instead of t_1. By the definition of $<_t$, $\lambda(t_1) = \lambda(t_2)$, and by the definition of T_b, $t_1 \bowtie t_2$. Hence, it follows that t_1 and t_2 are synthesised to atomic actions the same choice context. Therefore, provided the transition ordering $<_t$ is taken into account when sorting subexpressions in SORT (line 11 of SYNTHESISE), the ordering of the atomic actions corresponding to t_1 and t_2 will change as the ordering of t_1 and t_2 changes. Note that in SORT, $<_t$ will only come into play when comparing syntactically equivalent expressions. Hence,

this modification does not have any effect on the underlying expression that is synthesised.

There is some non-determinism in the construction of T' in line 6 of SCOPING. The definition of T' ensures that every transition $t' \in T'$ has a unique set of base transitions. Hence, the order in which transitions are considered as candidate members of T' will affect the contents of T'. However, it is only the size of T' that is important, and not the exact transitions contained in T'. Therefore, the choices in construction of T' have no effect on the form of the synthesised expression.

The order in which the transitions $x \in X$ are considered in INSERT SCOPING has an impact on the order in which new action names are used in the synthesised expression, and where several new actions are added at the same point, on the ordering of the new actions in the expression tree. The number of outcomes for the synthesised expression can be reduced by imposing a deterministic order on the transitions in X. One possibility is to associate a word with each transition $x \in X$, constructed by writing down in order the transitions in $T_b(x)$. The ordering could be defined by the location of the corresponding atomic action in the underlying expression tree (for example, an ordering based on a depth-first search of the expression tree would be suitable). Then, the lexicographic ordering of the words determines an ordering for the transitions in X. This ordering could also be used to provide a canonical choice for (t, l) in line 4 of INSERT SCOPING. Finally, a fixed order (for example, $n_1, n_2, n_3, ...$) for the new action names that are introduced in line 3 of INSERT SCOPING could be imposed.

Unfortunately, the choice of transition names in the input net can still have an effect on the location of new actions, due to the problem with the information lost about the structure of the input net when T_{sc} is removed. For example, Figure 4.24, shows the implementation of

$$E = ((a \parallel (\widehat{a}; b))\ \mathbf{sy}\ a) \parallel ((a \parallel (\widehat{a}; c))\ \mathbf{sy}\ a)$$

where the dotted transitions are those belonging to T_{sc}. Depending on the choice of ordering of transition names, the synthesised expression may be either of the following:

$$E = [\{n_1, n_2\} : (n_1 \,\Box\, a) \parallel (n_2 \,\Box\, a) \parallel ((\widehat{n_1} \,\Box\, \widehat{a}); b) \parallel ((\widehat{n_2} \,\Box\, \widehat{a}); c)]$$

$$E = [\{n_1, n_2\} : (n_1 \,\Box\, a) \parallel (n_2 \,\Box\, a) \parallel ((\widehat{n_2} \,\Box\, \widehat{a}); b) \parallel ((\widehat{n_1} \,\Box\, \widehat{a}); c)]$$

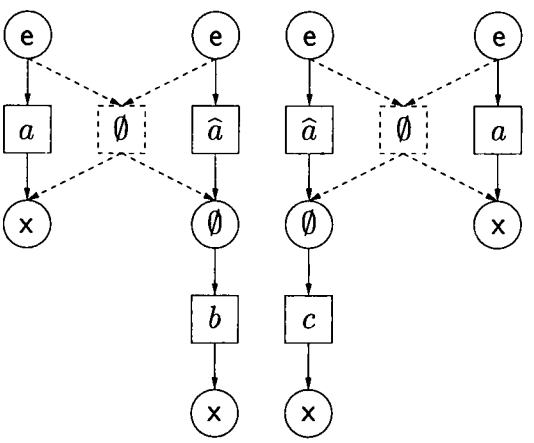


Figure 4.24: Non-determinism when adding new actions to represent transitions in T_{sc}

Summary

The investigation into the points of non-determinism of the synthesis algorithm presented in this chapter has shown that in order to generate canonical form expressions, some account of the structure, that the set of transitions T_{sc} provides to the input net, needs to be taken.

The main result is that if a canonical ordering for the transitions in the input net is available, and it is used, then the synthesis algorithm will provide a canonical form expression. In Section 4.6.3, this result is used to place some bounds on the time complexity of an algorithm for the syntax in Table 4.1, which synthesises canonical form expressions.

4.6.3 Bound on time complexity of canonical synthesis algorithm

In this section, the complexity of synthesising canonical form expressions is related to the complexity of GRAPH ISOMORPHISM, defined in Section 2.4.3.

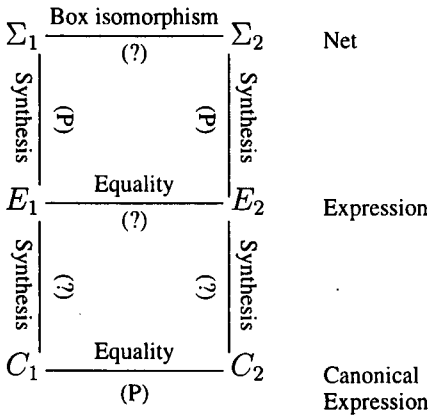


Figure 4.25: Relationship between the complexity of problems

Figure 4.25 gives a diagrammatic view of the relationship between the time complexity of the equivalence problem for nets, expressions, and canonical form expressions, for net semantic isomorphism.

- **Nets:** Let Σ_1, Σ_2 be implementations of expressions over the syntax in Table 4.1. The complexity of checking whether $\Sigma_1 =_{iso} \Sigma_2$ is clearly bounded by the complexity of GRAPH ISOMORPHISM. Note that it may be the case that the class of nets to which Σ_1 and Σ_2 is sufficiently restricted that the problem is easier than that of the generic GRAPH ISOMORPHISM problem (such as is the case for the basic Box expression syntax used in Chapter 3).

- **Expressions:** Suppose that Σ_1 and Σ_2 are synthesised to expressions E_1, E_2 over the syntax in Table 4.3. It has been shown that the synthesis can be completed in polynomial time in Section 4.6.1. Therefore, if the equality of E_1 and E_2, for net semantic isomorphism, can be compared

in time polynomial in the size of Σ_1 and Σ_2, then it follows that the comparison of Σ_1 and Σ_2 can be completed in polynomial time.

- **Canonical Expressions:** It is a trivial task to compare the equality of canonical form expressions C_1 and C_2. Therefore, using the same argument as above, if a polynomial time algorithm (based on the size of the nets Σ_1 and Σ_2) can be found to rewrite an expression into canonical form, then the tasks of comparing nets, and comparing expressions must also have polynomial time complexity.

Hence, the unknown time complexities, indicated by (?) in Figure 4.25 are all related. Based on the fact that it is not known whether an efficient algorithm for GRAPH ISOMORPHISM exists (*i.e.* it is not known whether GRAPH ISOMORPHISM is P or NP), it may be difficult to classify the unknown complexities in Figure 4.25.

In the following section, it is shown that for arbitrary expressions over the syntax in Table 4.3, the problem of checking the equality of expressions (or the nets that may be derived from them) for net semantic isomorphism, necessarily has the same time complexity as the generic GRAPH ISOMORPHISM problem.

Scoping equivalence and GRAPH ISOMORPHISM

In this section, it is shown that any solution to SCOPING EQUIVALENCE provides a solution to GRAPH ISOMORPHISM. This results places an upper bound on the time complexity of SCOPING EQUIVALENCE to be the time complexity of GRAPH ISOMORPHISM (It also places a lower bound on the complexity of GRAPH ISOMORPHISM

SCOPING EQUIVALENCE

INSTANCE: Expressions E_1, E_2 over the syntax in Table 4.3.

QUESTION: Is $E_1 =_{iso} E_2$?

(*i.e.* Are the implementations of E_1 and E_2 isomorphic?)

GRAPH ISOMORPHISM

INSTANCE: Graphs $G = (V, E)$, $G' = (V, E')$

QUESTION: Are G and G' "isomorphic", that is, is there a one-to-one function $f : V \to V$ such that $\{u, v\} \in E$ if and only if $\{f(u), f(v)\} \in E'$?

Let $G = (V, E)$, $G' = (V', E')$ be an arbitrary instance of GRAPH ISOMORPHISM. A corresponding instance of SCOPING EQUIVALENCE can be constructed from G and G'. Here, the construction of an expression E_1, from G is described. The construction of E_2 from G' follows an identical process.

Suppose $V = \{v_1, v_2, ..., v_n\}$ is the set of vertices in G. A base expression, E_b is constructed, which represents the vertices in G, but not the edges. Let $E_b = x_1 \parallel x_2 \parallel ... \parallel x_n$, where x_i is an action name corresponding to vertex v_i, for $1 \leq i \leq n$. The labelling function, μ, mapping action names to actions is such that $\mu(x_i) = \{a\}$ for $1 \leq i \leq n$. Each edge, $(v_j, v_k) \in E$ is represented by a synchronisation between the corresponding pair of actions in E_b. For example, suppose $(v_1, v_3) \in E$, then the representation of the edge could be added to E_b as follows:

$$[N : (x_1 \,\square\, n_1) \parallel x_2 \parallel (x_3 \,\square\, \widehat{n_1}) \parallel ... \parallel x_n]$$

where N is the set of new actions, that have been added to represent the synchronisation (*i.e.* $N = \{n_1\}$ in this case). Every edge in E may be represented in the same way – the only constraint is that a different "new" action is used to represent each edge.

It is clear that the construction of E_1 and E_2 from G and G' can be completed in polynomial time. Also, the form of expressions used to represent graphs are such that the size of the nets corresponding to these expressions are polynomial in the size of the expression. Therefore, the problem of check-

ing the isomorphism of nets Σ_1, Σ_2, which are implementations of expressions over the syntax in Chapter 4.3 necessarily has the same time complexity as the generic GRAPH ISOMORPHISM problem. Hence, the time complexity of SCOPING EQUIVALENCE is the same as the time complexity of GRAPH ISOMORPHISM.

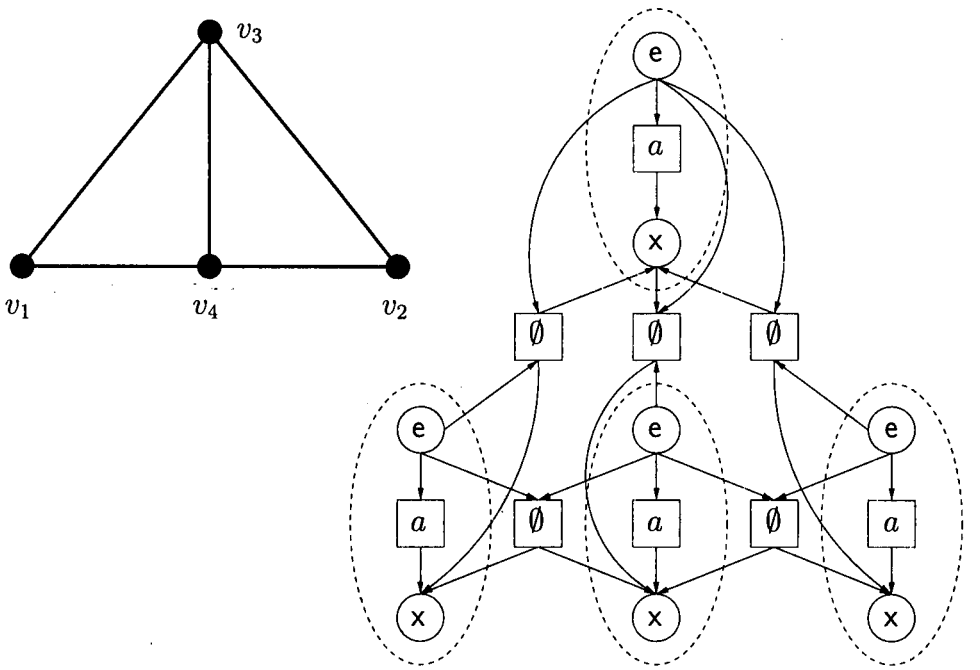


Figure 4.26: Construction of an expression from a graph

Figure 4.26 shows a graph, G, and the implementation of the expression E_1, constructed from G, where:

$$G = (\{v_1, v_2, v_3, v_4\}, \{(v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4), (v_3, v_4)\})$$

The base expression, representing the vertices of G is given by:

$$E_b = a \parallel a \parallel a \parallel a$$

The resulting expression, E_1, constructed from E_b by representing the set of edges in G is:

$$E = [\{n_1, n_2, n_3, n_4, n_5\} : (a \,\square\, n_1 \,\square\, n_2) \parallel (a \,\square\, n_3 \,\square\, n_4) \parallel (a \,\square\, \widehat{n_1} \,\square\, \widehat{n_3} \,\square\, n_5) \parallel (a \,\square\, \widehat{n_2} \,\square\, n_4 \,\square\, \widehat{n_5})]$$

Conclusion

The investigation into the time complexity and non-determinism of the synchronisation synthesis algorithm has not ruled out the possibility of an efficient algorithm that synthesises a canonical form expression:

- The proof of the equivalence of complexity of SCOPING EQUIVALENCE and GRAPH ISOMORPHISM uses a construction that is more expressive than can be derived from a expression over the syntax in Table 4.1. The semantics of the synchronisation operator place some quite severe restrictions on the structure of nets that may be represented, in comparison to the scoping operator. For example, it is possible to represent the structure of an arbitrary graph, but this is at the expense of every action representing a vertex of the graph having a unique label. Therefore, it is conceivable that the class of expressions over the syntax in Table 4.3 that can be derived from an expression over the syntax in Table 4.1 may be sufficiently constrained that the problems of checking equivalence and finding the canonical form become easier.

- The time complexity of GRAPH ISOMORPHISM has not been classified as either P or NP. It is an open problem whether an efficient algorithm exists for GRAPH ISOMORPHISM, and if one is found, the implication is that there is an efficient algorithm to synthesise canonical form expressions for nets derived from the syntax in Table 4.1.

The results of the investigation into the time complexity and non-determinism of the synthesis algorithm do not affect the possibility of the production of a sound and complete axiom system. However, these results do bear a relation to the time complexity of any proof strategy that would be used to apply the axioms and show the equivalence of a pair of expressions over the syntax in Table 4.1.

There are tools, such as *nauty* that use a heuristic approach to solve GRAPH ISOMORPHISM. Nauty also allows an arbitrary graph (or net!) to be relabelled

in a canonical form. Such a tool could be used to provide an algorithm that synthesises canonical form expressions, and also drive the application of axioms in a proof strategy.

4.6.4 Axiom system

In this section, an axiomatisation for the Box expression syntax in Table 4.1 is presented. The axiomatisation, like the synthesis algorithm, relies on the extra expressiveness provided by the scoping operator. Hence, the normal and canonical forms of expressions will use the syntax in Table 4.3 – *i.e.* all occurrences of the synchronisation operator will be rewritten in a form using the scoping operator. The axioms are presented in four groups:

- The axioms which provided the axiomatisation for the basic syntax in Chapter 3 are reused here.

- The axioms introduced in Proposition 35, which allow all scoping operators to be moved to immediately inside the enclosing iteration operator, or the top level of the expression, if there is no enclosing iteration operator.

- An axiom which allows the actions from the basic syntax, which are at the overlap between the basic syntax and synchronisation to be rewritten in scoping form, and vice versa.

- An axiom which allows all instances of the synchronisation operator to be rewritten in scoping form, and vice versa. For simplicity, this axiom is presented as a symmetric pair of rewriting rules that may be applied from left to right only.

Each of the axioms is shown to be sound. In Section 4.6.4, the axiom system is shown to be complete as well. Finally some examples of the application of the axiom system are presented in Section 4.6.5

The axioms relating to the basic syntax are shown below. The soundness for these axioms was discussed in Section 3.5.5 in Chapter 3, and will not be repeated here.

$$\begin{aligned}
(E_1; E_2); E_3 &= E_1; (E_2; E_3) \\
(E_1 \,\square\, E_2) \,\square\, E_3 &= E_1 \,\square\, (E_2 \,\square\, E_3) \\
(E_1 \parallel E_2) \parallel E_3 &= E_1 \parallel (E_2 \parallel E_3) \\
E_1 \,\square\, E_2 &= E_2 \,\square\, E_1 \\
E_1 \parallel E_2 &= E_2 \parallel E_1
\end{aligned}$$

The axioms which allow the positions of the scoping operators to be moved into the same positions used by the synthesis algorithm are given below. The soundness of these axioms follows directly from the semantics of the scoping operator.

$$\begin{aligned}
[N_1 : [N_2 : E]] &= [N_1 \cup N_2 : E] \text{ provided } N_1 \cap N_2 = \emptyset \\
[N : E_1] \parallel E_2 &= [N : E_1 \parallel E_2] \text{ provided } \forall n \in N : \mathcal{L}(E_2) \cap \{n, \widehat{n}\} = \emptyset \\
[N : E_1] \,\square\, E_2 &= [N : E_1 \,\square\, E_2] \text{ provided } \forall n \in N : \mathcal{L}(E_2) \cap \{n, \widehat{n}\} = \emptyset \\
E_1; [N : E_2] &= [N : E_1; E_2] \text{ provided } \forall n \in N : \mathcal{L}(E_1) \cap \{n, \widehat{n}\} = \emptyset \\
[N : E_1]; E_2 &= [N : E_1; E_2] \text{ provided } \forall n \in N : \mathcal{L}(E_2) \cap \{n, \widehat{n}\} = \emptyset
\end{aligned}$$

The following axiom allows certain atomic actions in the basic syntax to be rewritten in a form using the scoping operator. This axiom is more general than required to cope with the overlap between the basic syntax and synchronisation, but nevertheless does not cover all possible representations that are valid semantically.

$$\begin{aligned}
(E_1 \parallel E_2 \parallel ... \parallel (E_{k-1} \parallel E_k)...)) \,\square\, \alpha = [\{n_1, ..., n_{k-1}\} : (E_1 \,\square\, \widehat{n_1}) \parallel ((E_2 \,\square\, \widehat{n_2}) \parallel \\
... \parallel ((E_{k-1} \,\square\, \widehat{n_{k-1}}) \parallel (E_k \,\square\, (\{n_1, ..., n_{k-1}\} + \alpha)))..)]
\end{aligned}$$

When applying the axiom from left to right, note that each new basic action, n_i should not already be used elsewhere. The soundness of this axiom follows from the semantics of scoping, and from Proposition 27.

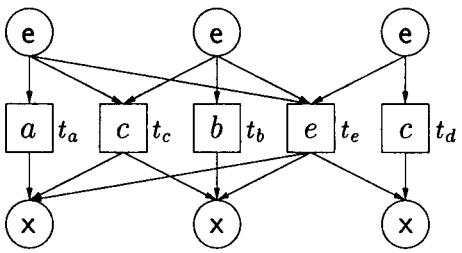


Figure 4.27: Choice axiom and multiple subexpressions

It is necessary for the axiom to consider multiple subexpressions, rather than a pair, because it needs to match the behaviour of INSERT SCOPING. For example, for an implementation of $E = (((a \parallel b) \Box c) \parallel d) \Box e$, shown in Figure 4.27, t_e satisfies the conditions for inclusion in T_{sc} (as does T_c), and $T_b(t_e) = \{t_a, t_b, t_d\}$. Therefore, the expression can be synthesised to (among others):

$$E' = [\{n_1, n_2, n_3\} : (a \Box \{n_1, c\} \Box \widehat{n_2}) \parallel (b \Box \widehat{n_1} \Box \widehat{n_3}) \parallel (d \Box \{n_2, n_3, e\})]$$

The subexpressions E_i for $1 \leq i \leq k$ in the axiom above can be restricted to be of the form $E_i' \Box x$ for some atomic action, x, without affecting the completeness of the axiomatisation. Since the axiom is sound in the form presented, the only effect of imposing the restriction is to complicate the axiom.

The definition of the semantics for synchronisation given in Section 1.3.5 is used in the rewriting rules which convert between synchronisation form and scoping form. The first rule converts the synchronisation operation in an expression to scoping form:

$$E \textbf{ sy } a \rightarrow [N' : E']$$

There is a condition in that E is required not to contain any synchronisation operations – *i.e.* the conversion from synchronisation to scoping form must be done in a bottom up fashion. The set of scoping actions, N', and the modified expression, E' are derived as follows:

Let A^a denote the set of action names, x, in E such that $\{a, \widehat{a}\} \cap \mu(x) \neq \emptyset$, and τ be a finite multiset of A^a. A function, f is defined, which expands

every action arising from a previous rewriting of a synchronisation operator in the multiset τ. Suppose N_{sc} is the set of basic actions appearing in scoping operations in E, A is the set of all action names in E, and for a multiset τ, define $X = \{x \in \tau \mid \mu(x) \cap N_{sc} = \emptyset\}$. The expanded multiset corresponding to τ is given by:

$$f(\tau) = \tau + \bigcup_{x \in (\tau - X)} (x' \in A \mid (\widehat{\mu(x')} \cap (\mu(x) \cap N_{sc})) \neq \emptyset)$$

The set of synchronisations and corresponding labels are given by:

$$\begin{aligned} A_{sy} &= \{f(\tau) \mid |\tau| \geq 2 \wedge \min(\Sigma_{x \in \tau} \mu(x)(a), \Sigma_{x \in \tau} \mu(x)(\widehat{a})) \geq |\tau| - 1\} \\ l(f(\tau)) &= ((\Sigma_{x \in \tau} \mu(x)) - ((|\tau| - 1) \cdot \{a, \widehat{a}\})) - N_{sc} \end{aligned}$$

A_{sy} and l are the equivalent action based synchronisations corresponding to the transition based synchronisation semantics given in Section 1.3. The use of the function f ensures that each $a_{sy} \in A_{sy}$ corresponds to the net based definition of base transitions for a synchronised transition. In fact, f does not find the exact base actions corresponding to the base transitions, but instead representatives from the same atomic choice contexts as the real base transitions.

The set of actions to scope by, N' (initially empty), and the modified expression, E' are constructed from E by applying the following process for each synchronisation $a_{sy} \in A_{sy}$:

- Construct $L = \{(x, \{\widehat{n_i}\}) \mid x \in a_{sy}$ and each n_i is a distinct new action$\}$.
- Choose any pair $(x, l) \in L$, and define $C = \{n \mid (x', \{\widehat{n}\}) \in L \wedge x' \neq x\}$. Add the set of new basic actions, C to N'. Replace (x, l) in L by $(x, C + l(a_{sy}))$.
- For each $(x, l) \in L$, replace x in E by $(x \,[\!]\, l)$.

Notice that this process follows the algorithm for INSERT SCOPING described in Section 4.4. The process is simpler when working on expressions, compared

to nets, because no account needs to be taken of the one to many mapping between actions in expressions and transitions in nets. The soundness of this rewriting rule follows from the semantics of synchronisation and scoping, and the correctness of INSERT SCOPING.

The second rewriting rule implements the reverse process of the first rule. Rather than a complex procedure to construct the resulting expression, there is a complex precondition that must hold before the rule can be applied. The precondition ensures that there is a representative in scoping form for every synchronisation that would be created by the application of the synchronisation operator.

$$[N : E] \rightarrow E' \textbf{ sy } a$$

A similar condition to the first rewriting rule is required, in that E must not contain any applications of the synchronisation operator – *i.e.* the conversion from scoping to synchronisation form must be done in a top down fashion.

E' is constructed from E by removing every atomic action α, such that $N \cap \alpha \neq \emptyset$. An intuitive notion of the construction of E' from E is given by the fact that $E' = E \textbf{ rs } N$. The precondition to the application of this rewriting rule is that the first rewriting rule may be applied to $E' \textbf{ sy } a$ to produce $[N : E]$.

Formally, and eliminating the need to make the correct choices at the points of non-determinism, the precondition may be checked by the following process:

- Construct E' from E, as described above, and set $N' = N$.
- Find the set of synchronisations, and corresponding labels, by applying the definitions for A_{sy} and l to $E' \textbf{ sy } a$.
- For each synchronisation $a_{sy} \in A_{sy}$, suppose $a_{sy} = \{x_1, x_2, ..., x_n\}$ and find a set of actions $x'_1, x'_2, ..., x'_n$ in E, such that for $1 \leq i \leq n$:
  - $\mu(x'_i) \cap (N' \cup \widehat{N'}) \neq \emptyset$
  - x'_i is in an atomic choice context with x_i

- $\exists 1 \leq j \leq n$ such that $\mu(x'_j) \cap \widehat{N'} \neq \emptyset$
- $\mu(x'_j) \cap N' = \bigcup_{1 \leq i \leq n, i \neq j} \widehat{\mu(x'_i)}$

If there is no set of actions meeting the conditions above, then the rewriting rule may not be applied. Otherwise, remove $\mu(x_j) \cap N'$ from N', and continue with the next element of A_{sy}.

- When all synchronisations in A_{sy} have been checked, the rewriting rule may be applied if and only if $N' = \emptyset$.

The soundness of this rewriting rule follows from the soundness of the first rewriting rule, and the associativity and commutativity of the choice operator.

Completeness

Let $\Sigma = (S, T, W, \lambda)$ be an implementation of an expression, E, over the syntax in Table 4.1. There are many possible output expressions from the synthesis algorithm, given input Σ, due to the points of non-determinism in the synthesis algorithm. The idea of the completeness proof presented in this section is to show that for any member, E', of the class of output expressions, E can be rewritten as E', using the axioms introduced in the previous section.

Proposition 37 *Let E be an expression over the syntax in Table 4.1, and E' be the output of the synthesis algorithm, when given an implementation of E as input. It is possible to rewrite E into E', using the axioms of Section 4.6.4.*

Proof: The aim of the proof is to show that any expression produced by the the synthesis algorithm can also be obtained using the axiom system. In order to do this, two areas need to be addressed. Firstly, the synthesis algorithm proceeds in a different order to the way in which axioms are intended to be applied. The synthesis algorithm proceeds by the following two steps:

- Synthesise the underlying expression, and rearrange the order of subexpressions.

- Add new actions and scoping operators into the underlying expression to produce the final synthesised expression.

In comparison, the strategy for applying the axioms of Section 4.6.4 is as follows:

- Convert all synchronisation operators, and those actions arising from the basic syntax, which overlap with the expressiveness of the synchronisation operator, to scoping form.
- Move all the scoping operators outwards as far as possible (*i.e.* until an iteration operator, or the outermost expression is reached.)
- Rearrange the order of the subexpressions.

Secondly, for each point of non-determinism in the synthesis algorithm, as analysed in Section 4.6.2, it is necessary to show that the possible outputs associated with the non-determinism can be generated using the axiomatisation.

Consider an arbitrary expression E over the syntax in Table 4.1, with Σ being an implementation of E. The underlying expression produced by the synthesis algorithm, given input Σ is effectively E with all synchronisation operators and some atomic actions in choice context removed, and the order of subexpressions in E rearranged. Bearing this in mind, the representation of the set of transitions, T_{sc} is treated first. The axiom which converts a synchronisation operator to scoping form follows the procedure INSERT SCOPING. All that needs to be shown is that for a synchronisation, τ, corresponding to a transition $t \in T_{sy}$, the expanded multiset of actions, $f(\tau)$ corresponds to $T_b(t)$.

It is valid to consider τ in terms of actions in an expression rather than the normal transitions in a net because E does not contain any synchronisation operators. When every action $x \in \tau$ is such that $\mu(x) \cap N_{sc} = \emptyset$ (*i.e.* x has not arisen from a previous conversion from synchronisation

to scoping form), τ matches $T_b(t)$ exactly. Note that if τ were used in place of $f(\tau)$, the axiom would still be sound – however, the form of expression produced would not match that of the synthesis algorithm.

The expanded multiset of actions is necessary to deal with actions that take part in a scoping operation. Suppose τ contains an action x such that $\mu(x) \cap N_{sc} \neq \emptyset$. Without loss of generality it may be assumed that $\mu(x)$ is of the form $\{n_{i1}, ..., n_{ik}, a_{j1}, ..., a_{jm}\}$, where $n_{ig} \in N_{sc}$ for $1 \leq g \leq k$, and $a_{jh} \notin N_{sc}$ for $1 \leq h \leq m$. Furthermore, there are actions $x_{i1}, ..., x_{ik}$ in E such that $\mu(x_{ig}) = \{\widehat{n_{ig}}\}$ for $1 \leq g \leq k$. In other words, the set of actions $A_1 = x_{i1}, ..., x_{ik}, x$ is a scoping form representation of a synchronisation. The expanded synchronisation, $f(\tau)$ replaces x by the set of actions, A_1. By the definition of INSERT SCOPING and the synchronisation to scoping form axiom, it can be seen that the actions in A_1 are in atomic choice contexts with the actual actions that were synchronised. Hence, in terms of choice contexts (in expressions) and connectivity (in nets), the multiset $f(\tau)$ matches $T_b(t)$.

The axiom for rewriting actions in atomic choice contexts into scoping form is more flexible than is required for rewriting the set of actions corresponding to T_{at}. No expansion, similar to that needed for synchronisation, is required here. However, in the synthesis algorithm an atomic choice transition may be included in T_{at} by virtue of another transition arising from synchronisation – for example in an implementation of $((a \parallel \{\widehat{a}, b\})$ **sy** $a \parallel \widehat{b}) \,\Box\, \emptyset$. If a transition arising from synchronisation is used in this way, then it is necessarily the case that the base transitions are all in the same choice context, and this is covered by the fact the axiom deals with an arbitrary number of subexpressions in the choice context (rather than a pair of subexpressions).

INSERT SCOPING adds the scoping operator at the highest possible point in the expression tree. In comparison, the application of the synchroni-

sation to scoping axiom places the scoping operator at the same point as the synchronisation operator it replaces. By definition, the basic actions which are scoped only appear within the scope of the scoping operator. Therefore, by the set of scoping axioms, and Proposition 35, the axiomatisation can be used to move the scoping operators into the same positions that would be obtained from the synthesis algorithm.

All the points of non-determinism in INSERT SCOPING, such as the choice of new action names, and the choice of the distinguished element of L are present in the corresponding axiom. Therefore, the same class of expressions resulting from these points of non-determinism can be obtained from both the axiomatisation and the synthesis algorithm.

The remaining point to deal with is the positions that new actions are added by calls to ADD. So far, it has been show that the axiomatisation will not necessarily add the actions in exactly the same location, but only in the same atomic choice context. However, the associativity and commutativity axioms for choice allow an arbitrary reordering of subexpressions in a choice context. A similar argument applies to the rearrangement of subexpressions to match the order in the underlying expression produced by the synthesis algorithm. Note that when all the scoping operators have been moved to match the locations used by the synthesis algorithm (*i.e.* immediately with iteration (sub)expressions, and surrounding the entire expression), then there are no constraints on the reordering of choice and parallel composition contexts, and on the bracketing of choice, sequence, and parallel composition contexts. Therefore, for any output expression E' from the synthesis algorithm on input Σ, the axiom system can be used to rewrite E as E', where Σ is an implementation of E. □

Theorem 5 *The axiom system presented in Section 4.6.4 is complete.*

Proof: Let E_1 and E_2 be two expressions over the syntax in Table 4.1, and Σ_1 and Σ_2 be implementations of E_1 and E_2 respectively. Suppose $\Sigma_1 =_{iso} \Sigma_2$, then it must be shown that E_1 may be rewritten as E_2 using the axiom system.

Given Σ_1 as input to the synthesis algorithm, any one of the expressions in the class E_C may be produced as output, due to the non-determinism in the synthesis algorithm. By Proposition 37, E_1 may be rewritten, using the axiomatisation, into any member of E_C. Since $\Sigma_2 =_{iso} \Sigma_1$, the output of the synthesis algorithm on input Σ_2 will be some member of E_C. Therefore E_2 may be rewritten, using the axiomatisation, into any member of E_C.

Suppose E is some member of E_C, then by applications of the axioms, both E_1 and E_2 may be rewritten as E. Therefore, E_1 may be rewritten as E_2 by concatenating the proof $E_1 = E$ with the proof that $E_2 = E$ written in reverse order (The synchronisation to scoping axiom may only be applied in one direction. However, the corresponding scoping to synchronisation axiom allows rewriting in the reverse direction). □

4.6.5 Examples

In this section, some properties of the synchronisation operator are demonstrated using the axiom system of Section 4.6.4. A list of all the axioms is presented in Table 4.12.

The axiomatisation consists of axioms (BS1-BS5) dealing with ordering and bracketing of subexpressions from the basic syntax, axioms (SP1-SP5) relating to the position of scoping operators, axiom CS1 for rewriting actions in atomic choice contexts in scoping form, and rewriting rules (SS1-SS2) dealing with the conversion between synchronisation and scoping forms. The axioms marked by "*" have preconditions which must be satisfied before the axiom can be applied – see Section 4.6.4. The procedure to apply the rewriting rules SS1

Axiom	Name
$(E_1; E_2); E_3 = E_1; (E_2; E_3)$	BS1
$(E_1 \,\square\, E_2) \,\square\, E_3 = E_1 \,\square\, (E_2 \,\square\, E_3)$	BS2
$(E_1 \parallel E_2) \parallel E_3 = E_1 \parallel (E_2 \parallel E_3)$	BS3
$E_1 \,\square\, E_2 = E_2 \,\square\, E_1$	BS4
$E_1 \parallel E_2 = E_2 \parallel E_1$	BS5
$[N_1 : [N_2 : E]] = [N_1 \cup N_2 : E]$	SP1*
$[N : E_1] \parallel E_2 = [N : E_1 \parallel E_2]$	SP2*
$[N : E_1] \,\square\, E_2 = [N : E_1 \,\square\, E_2]$	SP3*
$E_1; [N : E_2] = [N : E_1; E_2]$	SP4*
$[N : E_1]; E_2 = [N : E_1; E_2]$	SP5*
$(E_1 \parallel E_2 \parallel ... \parallel (E_{k-1} \parallel E_k)...)) \,\square\, \alpha =$ $[\{n_1, ..., n_{k-1}\} : (E_1 \,\square\, \widehat{n_1}) \parallel ((E_2 \,\square\, \widehat{n_2}) \parallel ... \parallel$ $((E_{k-1} \,\square\, \widehat{n_{k-1}}) \parallel (E_k \,\square\, (\{n_1, ..., n_{k-1}\} + \alpha)))..)]$	CS1
E **sy** $a \rightarrow [N' : E']$	SS1*
$[N : E] \rightarrow E'$ **sy** a	SS2*

Table 4.12: Axiomatisation of Box expression syntax in Table 4.1

and SS2 may also be found in Section 4.6.4.

Equivalent synchronisations

In this section, the equivalence of the following expressions is shown, using the axioms of Table 4.12:

$$
\begin{aligned}
E_1 &= ((a \parallel \widehat{b}) \parallel (\widehat{a} \,\square\, b)) \textbf{ sy } a \\
E_2 &= ((a \parallel \widehat{b}) \parallel (\widehat{a} \,\square\, b)) \textbf{ sy } b \\
E_3 &= ((a \parallel \widehat{b}) \parallel (\widehat{a} \,\square\, b)) \,\square\, \emptyset
\end{aligned}
$$

The implementation of the (equivalent) expressions is shown in Figure 4.28.

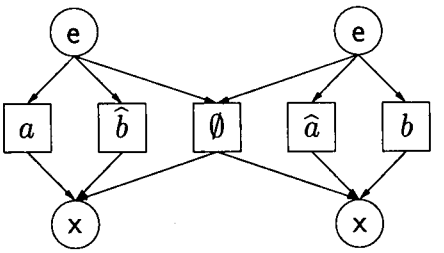


Figure 4.28: Implementation of E_1, E_2, and E_3

$$
\begin{aligned}
E_1 &= ((a \parallel \widehat{b}) \parallel (\widehat{a} \,\square\, b)) \textbf{ sy } a \\
&= [\{n_1\} : (a \,\square\, \widehat{n_1} \,\square\, \widehat{b}) \parallel (\widehat{a} \,\square\, n_1 \,\square\, b)] && \text{(SS1)} \\
&= ((a \parallel \widehat{b}) \parallel (\widehat{a} \,\square\, b)) \textbf{ sy } b && \text{(SS2)} \\
&= E_2 \\
&= [\{n_1\} : (a \,\square\, \widehat{b} \,\square\, \widehat{n_1}) \parallel (\widehat{a} \,\square\, b \,\square\, n_1)] && \text{(SS1)} \\
&= ((a \parallel \widehat{b}) \parallel (\widehat{a} \,\square\, b)) \,\square\, \emptyset && \text{(CS1)} \\
&= E_3
\end{aligned}
$$

Order of synchronisation

In this section, the equivalence of the following expressions is shown, using the axioms of Table 4.12:

$$
\begin{aligned}
E_1 &= ((a \parallel \{\widehat{a}b\}) \textbf{ sy } a \parallel \widehat{b}) \textbf{ sy } b \\
E_2 &= ((\{\widehat{a}b\} \parallel \widehat{b}) \textbf{ sy } b \parallel a) \textbf{ sy } a \\
E_3 &= (((a \parallel \{\widehat{a}b\}) \,\square\, b) \parallel \widehat{b}) \textbf{ sy } b \\
E_4 &= ((((\{\widehat{a}b\} \parallel \widehat{b}) \,\square\, \widehat{a}) \parallel a) \textbf{ sy } a
\end{aligned}
$$

The implementation of the (equivalent) expressions is shown in Figure 4.29. The first proof demonstrates that $E_1 = E_2$.

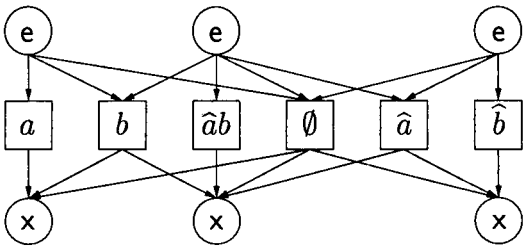


Figure 4.29: Implementation of E_1, E_2, E_3 and E_4

$$
\begin{aligned}
E_1 &= ((a \parallel \{\widehat{ab}\})\ \mathbf{sy}\ a \parallel \widehat{b})\ \mathbf{sy}\ b \\
&= ([\{n_1\} : (a \,\square\, \widehat{n_1}) \parallel (\{\widehat{ab}\} \,\square\, \{n_1 b\})] \parallel \widehat{b})\ \mathbf{sy}\ b && \text{(SS1)} \\
&= [\{n_2, n_3, n_4\} : [\{n_1\} : (a \,\square\, \widehat{n_1} \,\square\, \widehat{n_2}) \parallel \\
&\quad (\{\widehat{ab}\} \,\square\, \widehat{n_4} \,\square\, \{n_1 b\} \,\square\, \widehat{n_3})] \parallel (\widehat{b} \,\square\, \{n_4 \widehat{a}\} \,\square\, \{n_2 n_3\})] && \text{(SS1)} \\
&= [\{n_1, n_2, n_3, n_4 : ((a \,\square\, \widehat{n_1} \,\square\, \widehat{n_2}) \parallel (\{\widehat{ab}\} \,\square\, \widehat{n_4} \,\square\, \\
&\quad \{n_1 b\} \,\square\, \widehat{n_3})) \parallel (\widehat{b} \,\square\, \{n_4 \widehat{a}\} \,\square\, \{n_2 n_3\})] && \text{(SP1)} \\
&= [\{n_1, n_2, n_3, n_4 : (a \,\square\, \widehat{n_1} \,\square\, \widehat{n_2}) \parallel \\
&\quad ((\{\widehat{ab}\} \,\square\, \widehat{n_4} \,\square\, \{n_1 b\} \,\square\, \widehat{n_3}) \parallel (\widehat{b} \,\square\, \{n_4 \widehat{a}\} \,\square\, \{n_2 n_3\}))] && \text{(BS3)} \\
&= [\{n_1, n_2, n_3, n_4 : ((\{\widehat{ab}\} \,\square\, \widehat{n_4} \,\square\, \{n_1 b\} \,\square\, \widehat{n_3}) \parallel \\
&\quad (\widehat{b} \,\square\, \{n_4 \widehat{a}\} \,\square\, \{n_2 n_3\})) \parallel (a \,\square\, \widehat{n_1} \,\square\, \widehat{n_2})] && \text{(BS5)} \\
&= [\{n_1, n_2, n_3\} : [n_4 : ((\{\widehat{ab}\} \,\square\, \widehat{n_4} \,\square\, \{n_1 b\} \,\square\, \widehat{n_3}) \parallel \\
&\quad (\widehat{b} \,\square\, \{n_4 \widehat{a}\} \,\square\, \{n_2 n_3\})) \parallel (a \,\square\, \widehat{n_1} \,\square\, \widehat{n_2})]] && \text{(SP1)} \\
&= [n_4 : ((\{\widehat{ab}\} \,\square\, \widehat{n_4}) \parallel (\widehat{b} \,\square\, \{n_4 \widehat{a}\})) \parallel a]\ \mathbf{sy}\ a && \text{(SS2)} \\
&= ([n_4 : ((\{\widehat{ab}\} \,\square\, \widehat{n_4}) \parallel (\widehat{b} \,\square\, \{n_4 \widehat{a}\}))] \parallel a)\ \mathbf{sy}\ a && \text{(SP2)} \\
&= ((\{\widehat{ab}\} \parallel \widehat{b})\ \mathbf{sy}\ b \parallel a)\ \mathbf{sy}\ a && \text{(SS2)} \\
&= E_2
\end{aligned}
$$

From the above proof, $E_1 = ([\{n_1\} : (a \,\square\, \widehat{n_1}) \parallel (\{\widehat{ab}\} \,\square\, \{n_1 b\})] \parallel \widehat{b})\ \mathbf{sy}\ b$. Therefore:

$$
\begin{aligned}
E_1 &= ([\{n_1\} : (a \,\square\, \widehat{n_1}) \parallel (\{\widehat{ab}\} \,\square\, \{n_1 b\})] \parallel \widehat{b})\ \mathbf{sy}\ b \\
&= (((a \parallel \{\widehat{ab}\}) \,\square\, b) \parallel \widehat{b})\ \mathbf{sy}\ b && \text{(CS1)} \\
&= E_3
\end{aligned}
$$

Similarly, $E_1 = ([n_4 : ((\{\widehat{ab}\} \,\square\, \widehat{n_4}) \parallel (\widehat{b} \,\square\, \{n_4 \widehat{a}\}))] \parallel a)\ \mathbf{sy}\ a$ from the above

proof. Therefore:

$$\begin{aligned} E_1 &= ([n_4 : (((\{\widehat{ab}\} \,\Box\, \widehat{n_4}) \parallel (\widehat{b} \,\Box\, \{n_4\widehat{a}\}))] \parallel a) \textbf{ sy } a \\ &= ((((\{\widehat{ab}\} \parallel \widehat{b}) \,\Box\, \widehat{a}) \parallel a) \textbf{ sy } a \qquad \text{(CS1)} \\ &= E_4 \end{aligned}$$

Hence, it has been shown that $E_1 = E_2 = E_3 = E_4$.

Partially equivalent synchronisations

In this section, the equivalence of the following expressions is shown, using the axioms of Table 4.12:

$$\begin{aligned} E_1 &= (((a \,\Box\, b) \parallel (\widehat{a} \,\Box\, \widehat{c})); ((a \,\Box\, c) \parallel (\widehat{a} \,\Box\, \widehat{b}))) \textbf{ sy } a \\ E_2 &= ((((a \,\Box\, b) \parallel (\widehat{a} \,\Box\, \widehat{c})) \,\Box\, \emptyset); (((a \,\Box\, c) \parallel (\widehat{a} \,\Box\, \widehat{b})) \,\Box\, \emptyset)) \textbf{ sy } b \textbf{ sy } c \end{aligned}$$

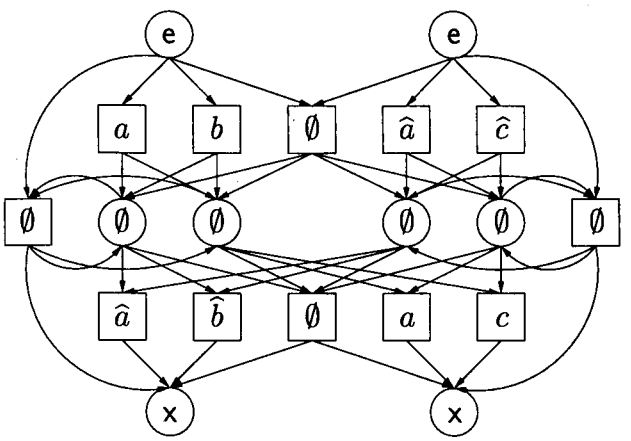


Figure 4.30: Implementation of E_1, and E_2

The implementation of E_1 and E_2 is shown in Figure 4.30.

$$
\begin{array}{lll}
E_1 & = & (((a \,\square\, b) \parallel (\widehat{a} \,\square\, \widehat{c})); ((a \,\square\, c) \parallel (\widehat{a} \,\square\, \widehat{b}))) \textbf{ sy } a \\
& = & [\{n_1, n_2, n_3, n_4\} : ((a \,\square\, \widehat{n_1} \,\square\, n_2 \,\square\, b) \parallel (\widehat{a} \,\square\, n_1 \,\square\, \widehat{n_3} \,\square\, \widehat{c})); \\
& & ((a \,\square\, n_3 \,\square\, \widehat{n_4} \,\square\, c) \parallel (\widehat{a} \,\square\, \widehat{n_2} \,\square\, n_4 \,\square\, \widehat{b}))] \qquad \text{(SS1)} \\
& = & [n_3 : [\{n_1, n_2, n_4\} : ((a \,\square\, \widehat{n_1} \,\square\, n_2 \,\square\, b) \parallel (\widehat{a} \,\square\, n_1 \,\square\, \widehat{n_3} \,\square\, \widehat{c})); \\
& & ((a \,\square\, n_3 \,\square\, \widehat{n_4} \,\square\, c) \parallel (\widehat{a} \,\square\, \widehat{n_2} \,\square\, n_4 \,\square\, \widehat{b}))]] \qquad \text{(SP1)} \\
& = & [\{n_1, n_2, n_4\} : ((a \,\square\, \widehat{n_1} \,\square\, n_2 \,\square\, b) \parallel (\widehat{a} \,\square\, n_1 \,\square\, \widehat{c})); \\
& & ((a \,\square\, \widehat{n_4} \,\square\, c) \parallel (\widehat{a} \,\square\, \widehat{n_2} \,\square\, n_4 \,\square\, \widehat{b}))] \textbf{ sy } c \qquad \text{(SS2)} \\
& = & [n_2 : [\{n_1, n_4\} : ((a \,\square\, \widehat{n_1} \,\square\, n_2 \,\square\, b) \parallel (\widehat{a} \,\square\, n_1 \,\square\, \widehat{c})); \\
& & ((a \,\square\, \widehat{n_4} \,\square\, c) \parallel (\widehat{a} \,\square\, \widehat{n_2} \,\square\, n_4 \,\square\, \widehat{b}))]] \textbf{ sy } c \qquad \text{(SP1)} \\
& = & [\{n_1, n_4\} : ((a \,\square\, \widehat{n_1} \,\square\, b) \parallel (\widehat{a} \,\square\, n_1 \,\square\, \widehat{c})); \\
& & ((a \,\square\, \widehat{n_4} \,\square\, c) \parallel (\widehat{a} \,\square\, n_4 \,\square\, \widehat{b}))] \textbf{ sy } b \textbf{ sy } c \qquad \text{(SS2)} \\
& = & [n_4 : [n_1 : ((a \,\square\, \widehat{n_1} \,\square\, b) \parallel (\widehat{a} \,\square\, n_1 \,\square\, \widehat{c})); \\
& & ((a \,\square\, \widehat{n_4} \,\square\, c) \parallel (\widehat{a} \,\square\, n_4 \,\square\, \widehat{b}))]] \textbf{ sy } b \textbf{ sy } c \qquad \text{(SP1)} \\
& = & [n_4 : [n_1 : ((a \,\square\, \widehat{n_1} \,\square\, b) \parallel (\widehat{a} \,\square\, n_1 \,\square\, \widehat{c}))]; \\
& & ((a \,\square\, \widehat{n_4} \,\square\, c) \parallel (\widehat{a} \,\square\, n_4 \,\square\, \widehat{b}))] \textbf{ sy } b \textbf{ sy } c \qquad \text{(SP5)} \\
& = & [n_1 : ((a \,\square\, \widehat{n_1} \,\square\, b) \parallel (\widehat{a} \,\square\, n_1 \,\square\, \widehat{c}))]; \\
& & [n_4 : ((a \,\square\, \widehat{n_4} \,\square\, c) \parallel (\widehat{a} \,\square\, n_4 \,\square\, \widehat{b}))] \textbf{ sy } b \textbf{ sy } c \qquad \text{(SP4)} \\
& = & [n_1 : ((a \,\square\, b \,\square\, \widehat{n_1}) \parallel (\widehat{a} \,\square\, \widehat{c} \,\square\, n_1)]; \\
& & [n_4 : ((a \,\square\, c \,\square\, \widehat{n_4}) \parallel (\widehat{a} \,\square\, \widehat{b} \,\square\, n_4))] \textbf{ sy } b \textbf{ sy } c \qquad \text{(BS2)} \\
& & \qquad \text{(BS4)} \\
& = & (((a \,\square\, b) \parallel (\widehat{a} \,\square\, \widehat{c}) \,\square\, \emptyset); \\
& & [n_4 : ((a \,\square\, c \,\square\, \widehat{n_4}) \parallel (\widehat{a} \,\square\, \widehat{b} \,\square\, n_4))] \textbf{ sy } b \textbf{ sy } c \qquad \text{(CS1)} \\
& = & (((a \,\square\, b) \parallel (\widehat{a} \,\square\, \widehat{c}) \,\square\, \emptyset); (((a \,\square\, c) \parallel (\widehat{a} \,\square\, \widehat{b})) \,\square\, \emptyset)) \textbf{ sy } b \textbf{ sy } c \qquad \text{(CS1)} \\
& = & E_2
\end{array}
$$

Chapter 5

Duplication Equivalence

In this chapter, the synthesis and axiomatisation problems are investigated for net semantic duplication equivalence. When duplication equivalence is used for identifying nets, the synthesis problem, for input net, Σ, which is duplication equivalent to the implementation of some unknown box expression, is to find a box expression, E, such that Σ is duplication equivalent to the implementation of E. The important difference between isomorphism and duplication equivalence is that the input net is not necessarily an implementation of a box expression. For example, for the the basic syntax described by Table 2.3 in Chapter 2, the net shown in Figure 5.1 (i) is an implementation of $E = a$, while there is no expression whose implementation is net (ii) in Figure 5.1. However, the nets in Figure 5.1 are duplication equivalent, and are both valid inputs to a synthesis algorithm where the net semantic used is duplication equivalence.

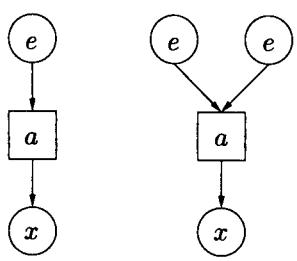


Figure 5.1: Duplication equivalent nets

The axiomatisation problem, for duplication equivalence, is to find a sound and complete set of axioms that characterise the notion of duplication equivalence of expressions. There are two approaches to constructing such an axiomatisation:

- Extend the axiomatisation for isomorphism. This approach relies on the fact that any pair of isomorphic nets are necessarily duplication equivalent (*i.e.* duplication equivalence encompasses isomorphism). A set of axioms that exactly characterises the differences between isomorphism and duplication equivalence need to be found. Such a set of axioms, together with the axiomatisation for isomorphism provide an axiomatisation for duplication equivalence.

- Construct a completely new axiomatisation. This approach is more flexible in that it does not require a subset of the axioms to characterise the differences between isomorphism and duplication equivalence. However, no advantage is taken of the work that has already been done in producing an axiomatisation for isomorphism.

Section 5.1 briefly discusses the synthesis and axiomatisation problems for the basic syntax, and the basic syntax with the synchronisation operator added. The following sections provide a more in-depth investigation into duplication equivalence. Section 5.2 extends the work in Chapter 2 from the domain of isomorphism to that of duplication equivalence; Section 5.3 gives an overview of how the investigation into synchronisation of Chapter 4 can be extended for duplication equivalence; and Section 5.4, motivated by the result of Proposition 22 constructs a completely new axiomatisation for the synchronisation operator, for net semantic duplication equivalence.

5.1 Extension from isomorphism to duplication equivalence

5.1.1 Basic syntax

The approach taken in investigating duplication equivalence for the basic syntax is to define a canonical form for input nets to the synthesis algorithm presented in Chapter 3. Let E be an expression from the basic syntax in Table 3.1, and Σ be the implementation of E. The input to the synthesis algorithm for net semantic duplication equivalence shall be Σ', constructed from Σ by removing all duplicate places and transitions.

This approach allows the results of Chapter 3 to be reused. However, there is an assumption here that Σ' is, in fact, the implementation of some expression over the basic syntax. The axiomatisation of Chapter 3 will be extended to allow the rewriting of E to E', where the implementation of E' is Σ'.

Section 5.2 demonstrates that this approach is possible and extends the synthesis algorithm and axiomatisation of Chapter 3 from net semantic isomorphism to duplication equivalence.

5.1.2 Synchronisation

In Chapter 4, Proposition 22 demonstrated that the synthesis problem for nets obtained from expressions over the syntax in Table 4.1 is NP hard. It is possible to recast this result in terms of the equivalence of expressions, which rules out any efficient proof strategy for an axiomatisation of the syntax in Table 4.1.

The analysis in Section 4.2.5 demonstrated that the NP hardness result no longer held when duplication equivalence was used in place of isomorphism. Hence there are two general approaches to finding an axiomatisation for duplication equivalence.

- Re-investigate the possibility of producing an axiomatisation purely in

the domain of the syntax in Table 4.1 – *i.e.* without moving to the scoping form representation for synchronisation.

- Extend the axiomatisation of Chapter 4, which rewrites synchronisation operations into scoping form, to capture the difference between isomorphism and duplication equivalence.

Section 5.4 briefly discusses the extension of the axiomatisation of Chapter 4, and presents a detailed investigation into an axiomatisation for duplication equivalence which stays with the domain of the syntax in Table 4.1, and does not resort to the use of a scoping representation.

5.2 Basic syntax

In order to use the synthesis algorithm and axiomatisation of Chapter 3 as a basis for an investigation into duplication equivalence, it is necessary to show that for any implementation, Σ, of an expression, E, the duplicate free version of Σ is an implementation of some expression from the basic syntax.

Proposition 1 shows that duplicated places never arise in the implementation of a basic syntax box expression. Therefore, only the duplication of transitions needs to be considered. Firstly, it is shown that it is not possible for the implementation of a basic syntax box expression to contain an isolated transition:

Proposition 38 *For any implementation, $\Sigma = (S, T, W, \lambda)$, of an expression from the syntax in Table 3.1, every transition $t \in T$ is such that ${}^\bullet t \neq \emptyset$ and $t^\bullet \neq \emptyset$.*

Proof: Follows directly from the semantics for the syntax in Table 3.1. □

Proposition 39 *Let E be an expression over the syntax in Table 3.1, such that E does not contain atomic actions with the same label in the same choice context. Let Σ be an implementation of E. Σ does not contain any duplicate transitions.*

Proof: By structural induction over the box expression syntax in Table 3.1.

Base case: $E = \alpha$: By definition, the implementation of α contains a single transition. Therefore no duplicate transitions are present.

Induction step: By the induction hypothesis, the implementation, $\Sigma_i = (S_i, T_i, W_i, \lambda_i)$ of subexpression, E_i, does not contain any duplicate transitions. Let Σ be an implementation of E, constructed from disjoint implementations of the subexpressions. Therefore, the only possibility for duplicate transitions, t_1, t_2 is where t_1 and t_2 arise from different subexpressions, E_j, E_k of E.

- $E = E_1 \parallel E_2$: Without loss of generality, consider transitions $t_1 \in T_1$ and $t_2 \in T_2$. By Proposition 38, t_1 and t_2 are not isolated transitions t_2. Therefore, by the semantics of parallel composition, t_1 cannot be a duplicate of t_2 because, t_1 and t_2 are in different disjoint subnets. Hence Σ does not contain any duplicate transitions.
- $E = E_1; E_2$: Without loss of generality, consider transitions $t_1 \in T_1$ and $t_2 \in T_2$. By Proposition 38, t_1 and t_2 are not isolated transitions. By the semantics of sequence, and Proposition 7 and 4, there exists a cluster of places, which when removed is such that $t_1 \overset{\leftrightarrow}{\sim}_{N_i} T_e(\Sigma)$ and $t_2 \overset{\leftrightarrow}{\sim}_{N_i} T_x(\Sigma)$, but for any $t_e \in T_e(\Sigma)$, $t_x \in T_x(\Sigma)$ there is no undirected path between t_e and t_x. Therefore, t_1 and t_2 are not duplicates of each other, and Σ does not contain any duplicate transitions.
- $E = E_1 \,[]\, E_2$: Without loss of generality, consider transitions $t_1 \in T_1$ and $t_2 \in T_2$. By the semantics of choice composition an Proposition 4, t_1 and t_2 cannot be duplicates if either transition is connected to any internal place. Suppose there exists $s_1 \in S_e(\Sigma_1)$ such that $s_1 \notin {}^\bullet t_1$, and there exists $s_2 \in S_e(\Sigma_2)$ such that $s_2 \in {}^\bullet t_2$. Then by the semantics of choice composition, there is an arc from $\{s_1, s_2\}$ to t_2 to t_1. Therefore t_1 is not a duplicate of t_2 if there is an entry

place $s_e \in S_1$ such that $s_e \notin {}^\bullet t_1$. A similar argument can be applied to the entry places of t_2, and to the exit places of both transitions. Now consider transitions $t_1 \in T_1$, $t_2 \in T_2$ such that:

$$\begin{aligned} {}^\bullet t_1 &= S_e(\Sigma_1) \\ t_1{}^\bullet &= S_x(\Sigma_1) \\ {}^\bullet t_2 &= S_e(\Sigma_2) \\ t_2{}^\bullet &= S_x(\Sigma_2) \end{aligned}$$

By the semantics of choice composition, then t_1 is a duplicate of t_2, provided $\lambda(t_1) = \lambda(t_2)$. By Proposition 27, the transitions t_1 and t_2 arise from atomic actions in the same choice context. However, E has been restricted not to contain atomic actions with the same label in the same choice context. Therefore, Σ does not contain any duplicate transitions.

- $[E_1 * E_2 * E_3]$: Recall that two copies of each of the subnets Σ_1, Σ_2 and Σ_3 are used in the construction of Σ. Denote the two copies of Σ_i by Σ_{i1} and Σ_{i2} for $1 \leq i \leq 3$. The following sources for the transitions t_1 and t_2 need to be considered (not all possible combinations need to be considered because of the symmetry of the semantics for iteration):

Number	Source of t_1	Source of t_2
1	Σ_{11}	Σ_{12}
2	Σ_{11}	Σ_{21}
3	Σ_{11}	Σ_{22}
4	Σ_{11}	Σ_{31}
5	Σ_{11}	Σ_{32}
6	Σ_{21}	Σ_{22}
7	Σ_{21}	Σ_{31}
8	Σ_{21}	Σ_{32}
9	Σ_{31}	Σ_{32}

A similar argument to that used for sequence composition can be used to show that t_1 and t_2 cannot be duplicates for cases 2-8. In case 1, it is not possible for t_1 and t_2 to share post places. Similarly, in case 9, it is necessarily the case that ${}^\bullet t_1 \neq {}^\bullet t_2$. Therefore, by Proposition 38, t_1 and t_2 are not duplicates in cases 1 and 9. Hence Σ does not contain any duplicate transitions.

□

Corollary 6 *Duplicate transitions may only arise in the basic syntax from atomic actions which have the same label, and are in the same choice context.*

Proof: Proposition 39 shows that duplicate transitions cannot arise in any other way. The fact that duplicate transitions may only arise from atomic actions which have the same label, and are in the same choice context follows from Proposition 27, and the semantics for choice composition. □

All of the results of this section follow from Corollary 6. For each of the areas investigated for isomorphism in Chapter 3, the corresponding result for duplication equivalence is presented here.

5.2.1 Synthesis Algorithm

The synthesis algorithm of Chapter 3 is modified so that a canonical form for the input net is constructed before the synthesis process takes place. The canonical form net is constructed by removing all duplicate places and transitions from the input net. By Proposition 1 and Corollary 6, it is guaranteed that if the input net is duplication equivalent to an implementation of a basic syntax box expression, then the canonical form net is isomorphic to the implementation of a basic syntax expression. The modified version of BOX EXPRESSION SYNTHESIS is given below.

BOX EXPRESSION SYNTHESIS(Σ)

1 N=*new* node

2 N.net=CANONICAL NET(Σ)

3 SYNTHESISE(N)

4 **return** EXPRESSION(N)

5.2.2 Time complexity of the Synthesis Algorithm

Let $\Sigma = (S, T, W, \lambda)$, and $N = S \cup T$. The time complexity of CANONICAL NET (*i.e.* identifying and removing all duplicate places and transitions) is $O(n^3)$, where $n = |N|$ is the number of nodes in the input net. A check can be made whether a pair of nodes, n_1 and n_2, duplicate each other in $O(n)$ time by comparing $W(n_1, n)$ and $W(n, n_1)$ with $W(n_2, n)$ and $W(n, n_2)$ for all $n \in N$. The time complexity of $O(n^3)$ is obtained from the time it takes to check all n^2 possible pairs of nodes.

Therefore, from the results in Section 3.5.1, the time complexity of BOX EXPRESSION SYNTHESIS is $O(n^5) + O(n^3)$ – *i.e.* the time complexity remains at $O(n^5)$.

5.2.3 Canonical Box Expression Synthesis

The CANONICAL BOX EXPRESSION SYNTHESIS algorithm can be modified in exactly the same way so that it is possible to synthesise canonical expressions in the domain of duplication equivalence. In a similar way to BOX EXPRESSION SYNTHESIS, the overall time complexity of the algorithm is not affected by these modifications.

CANONICAL BOX EXPRESSION SYNTHESIS(Σ)

1 N=*new* node

2 N.net=CANONICAL NET(Σ)

```
3   ORDERED SYNTHESISE(N)
4   return EXPRESSION(N)
```

5.2.4 Canonical Box Expression

In order to deal with duplicate transitions, the CANONICAL BOX EXPRESSION algorithm needs to be able to identify and remove identically labelled atomic actions appearing the the same choice context.

The definition of the ordered standard form of Section 3.5.3 guarantees that atomic actions giving rise to duplicate transitions will be adjacent in ordered expressions. Therefore, it is a simple task to remove such adjacent transitions in choice contexts in the expression. The modified code for CANONICAL BOX EXPRESSION is given below:

```
CANONICAL BOX EXPRESSION(E)
1   N=expression tree corresponding to standard form of E
2   VISIT(N)
3   return EXPRESSION(N)

VISIT(N)
1   if N.type≠atomic
2     for each node N' in N.list
3         do VISIT(N')
4     if N.type=parallel or choice
5       then sort(N.list)
6     if N.type=choice
7       then remove duplicates(N.list)
```

The time comlexity of removing duplicates is $O(a)$, where a is the number of atomic actions in the box expression, E. Therefore, the time complexity of the VISIT procedure remains at $O(a^2 \cdot loga)$.

5.2.5 Decision Problems

The decision problems PETRI BOX DUPLICATION EQUIVALENCE and BOX EXPRESSION DUPLICATION EQUIVALENCE can be solved using the modified versions of CANONICAL BOX EXPRESSION SYNTHESIS and CANONICAL BOX EXPRESSION presented in this section. The pseudo code, which is identical to the corresponding algorithms for isomorphism is shown below.

PETRI BOX DUPLICATION EQUIVALENCE(Σ_1, Σ_2)
1 C_1=CANONICAL BOX EXPRESSION SYNTHESIS (Σ_1)
2 C_2=CANONICAL BOX EXPRESSION SYNTHESIS(Σ_2)
3 **if** $C_1 = C_2$
4 **then return** yes
5 **else return** no

BOX EXPRESSION DUPLICATION EQUIVALENCE(E_1, E_2)
1 C_1=CANONICAL BOX EXPRESSION(E_1)
2 C_2=CANONICAL BOX EXPRESSION(E_2)
3 **if** $C_1 = C_2$
4 **then return** yes
5 **else return** no

When comparing atomic actions, α_1 and α_2 in canonical form expressions, C_1 and C_2, the words $\mathcal{A}(\alpha_1)$ and $\mathcal{A}(\alpha_2)$ should be compared.

5.2.6 Axiom system

Only one additional axiom is required above those in Table 3.6, namely $\alpha \,\square\, \alpha = \alpha$. This follows from the fact that subexpressions in the same choice context may be arbitrarily reordered using the axioms relating to associativity and commutativity of the choice operator. The complete axiom system is shown in Table 5.1.

Associativity	$(E_1; E_2); E_3 = E_1; (E_2; E_3)$
	$(E_1 \,[]\, E_2) \,[]\, E_3 = E_1 \,[]\, (E_2 \,[]\, E_3)$
	$(E_1 \parallel E_2) \parallel E_3 = E_1 \parallel (E_2 \parallel E_3)$
Commutativity	$E_1 \,[]\, E_2 = E_2 \,[]\, E_1$
	$E_1 \parallel E_2 = E_2 \parallel E_1$
Duplication	$\alpha \,[]\, \alpha = \alpha$

Table 5.1: Axioms

5.2.7 Generating Proofs

In this section, the CANONICAL PROOF algorithm of Section 3.5.6 is extended to provide a proof that an expression is equivalent to its canonical form for net semantic duplication equivalence.

The algorithm for CANONICAL PROOF can be modified by checking for adjacent atomic actions with the same label in choice contexts. Such atomic actions are guaranteed to be adjacent due to the fact that the choice context is rearranged by a call to SORT. The pseudo-code for BRACKET, SORT, and ORDER is not repeated here as no changes are required to these procedures. Recall that the variables Proof and T' are accessible globally, where Proof is a list of parse trees, and T' is a pointer to the root of the parse tree that is manipulated by the algorithm. The statement Proof=Proof+T' appends a copy of the parse tree, T' to Proof.

```
CANONICAL PROOF(E)
1   T'= parse tree of E
2   Proof=[T']
3   BRACKET(T')
4   SORT(T')
5   REMOVE DUPLICATES(T')
```

```
6   return Proof

REMOVE DUPLICATES(T)
1   case T.type
2       atomic: do nothing
3       iteration: REMOVE DUPLICATES(T.left)
4                  REMOVE DUPLICATES(T.middle)
5                  REMOVE DUPLICATES(T.right)
6       sequence,parallel:
7                  REMOVE DUPLICATES(T.left)
8                  REMOVE DUPLICATES(T.right)
9       choice:
10                 if T.left.type=atomic then
11                   case T.right.type
12                       atomic:
13                         if T.left.action=T.right.action then
14                           T=T.right
15                           Proof=Proof+T'
16                           REMOVE DUPLICATES(T)
17                       choice:
18                         if T.right.left.type=atomic then
19                           if T.left.action=T.right.left.action then
20                             temp=T.left
21                             T.left=T.right
22                             T.right=T.left.right
23                             T.left.right=temp
24                             Proof=Proof+T'
25                             T.left=T.left.right
26                             Proof=Proof+T'
27                             REMOVE DUPLICATES(T)
```

```
28                    parallel,sequence,iteration:
29                      REMOVE DUPLICATES(T.right)
30                  else
31                    REMOVE DUPLICATES(T.left)
32                    REMOVE DUPLICATES(T.right)
```

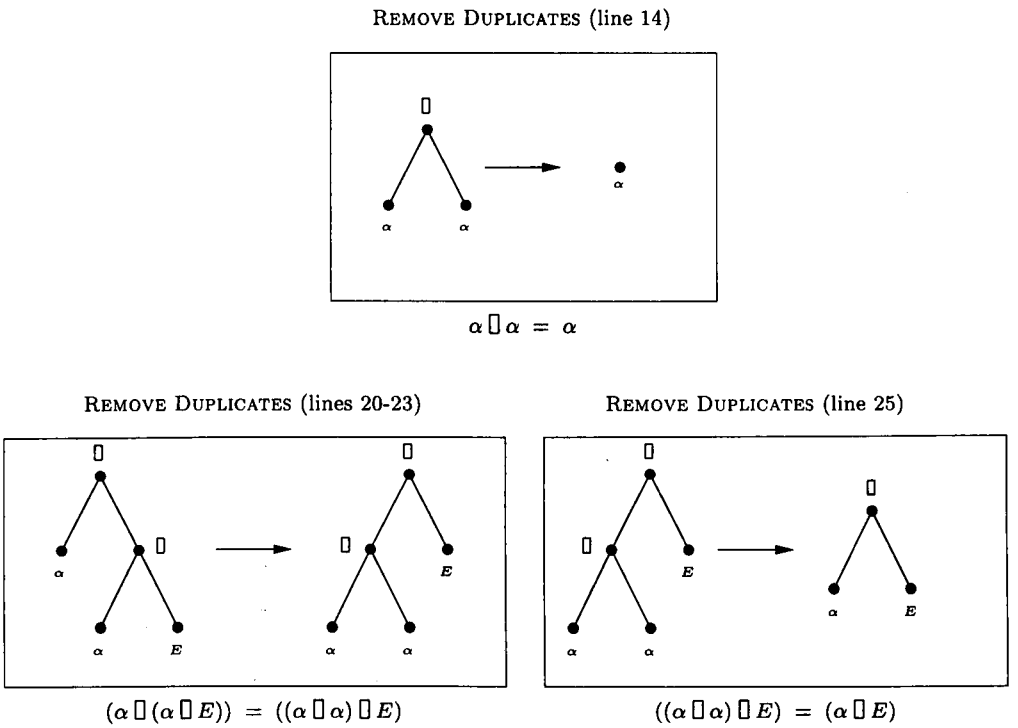


Figure 5.2: Manipulation of the parse tree

Figure 5.2 shows the tree manipulations carried out in line 14 and in lines 20-25 of REMOVE DUPLICATES, together with the corresponding expression manipulations. The manipulations in lines 14 and 25 correspond to a application of $\alpha \,\square\, \alpha = \alpha$, and the manipulation in lines 20-23 corresponds to an application of the associativity axiom for choice composition.

Let a be the number of atomic actions in a Box expression, E. The time complexity of the REMOVE DUPLICATES procedure is $O(a)$, and $O(a)$ axiom applications are performed.

Let E_1, E_2 be Box expressions such that $\mathsf{box}(E_1) = \mathsf{box}(E_2)$. A proof that

$E_1 = E_2$ can be generated using the proofs provided by CANONICAL PROOF as follows:

BOX EXPRESSION DUPLICATION EQUIVALENCE PROOF(E_1, E_2)

1 Proof$_1$=CANONICAL PROOF(E_1)

2 Proof$_2$=CANONICAL PROOF(E_2)

3 Output Proof$_1$

4 Output Proof$_2$ in reverse order.

The time complexity of the algorithm, on input E_1 and E_2 is $O(a^3)$, where $a = \max\{a_1, a_2\}$, and a_1 and a_2 are the number of atomic actions in E_1 and E_2 respectively. The length of the proof generated by BOX EXPRESSION DUPLICATION EQUIVALENCE PROOF is $O(a^3)$. In other words, the time complexity and bound on proof length is not affected by the extension from isomorphism to duplication equivalence.

5.2.8 Examples

Figure 5.3 shows nets Σ_1, duplication equivalent to an implementation of $E_1 = (a; b); (c \,[]\, c)$; Σ_2, duplication equivalent to an implementation of $E_2 = (a \,[]\, a); ((b \,[]\, (b \,[]\, b)); c)$; and Σ_3 the canonical form net for Σ_1 and Σ_2 by removing duplicate places and transitions. The synthesis algorithm produces the following outputs given Σ_1 and Σ_2 as input:

$$\text{BOX EXPRESSION SYNTHESIS}(\Sigma_1) = (a; b); c$$

$$\text{BOX EXPRESSION SYNTHESIS}(\Sigma_2) = a; (b; c)$$

Using CANONICAL BOX EXPRESSION SYNTHESIS, both input nets result in the synthesised expression $a; (b; c)$. The same result is obtained from CANONICAL BOX EXPRESSION on inputs E_1 and E_2. Hence, PETRI BOX

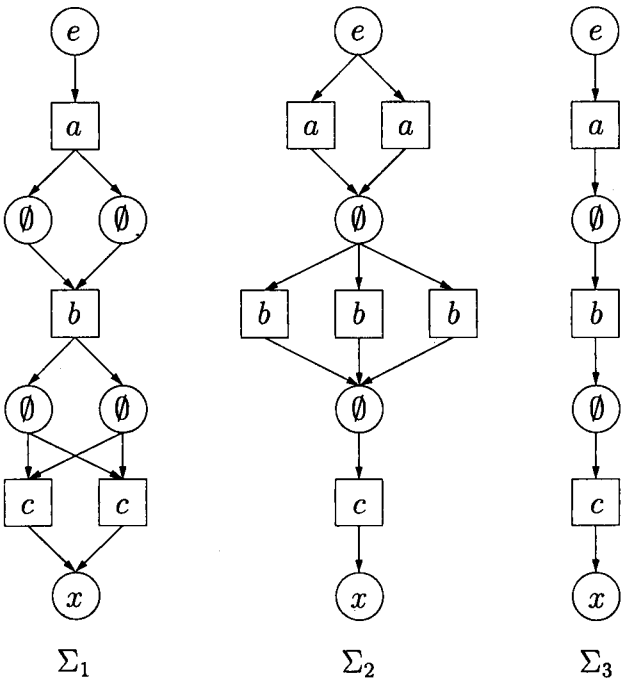


Figure 5.3: Example nets

DUPLICATION EQUIVALENCE(Σ_1, Σ_2), and BOX EXPRESSION DUPLICATION EQUIVALENCE(E_1, E_2) both produce the output "yes".

A call to BOX EXPRESSION DUPLICATION EQUIVALENCE PROOF(E_1, E_2) generates the following proof that E_1 and E_2 are equivalent (via their canonical forms C_1 and C_2):

$$\begin{aligned}
E_1 &= (a;b);(c \,\square\, c) \\
&= a;(b;(c \,\square\, c)) \\
&= a;(b;c) \;= C_1 \;=\; C_2 \\
&= a;((b \,\square\, b);c) \\
&= a;(((b \,\square\, b) \,\square\, b);c) \\
&= a;((b \,\square\, (b \,\square\, b));c) \\
&= (a \,\square\, a);((b \,\square\, (b \,\square\, b));c) \\
&= E_2
\end{aligned}$$

As with BOX EXPRESSION ISOMORPHISM PROOF, the proof will not generally

be the shortest possible. However, the generated proof has a length at most polynomial in the size of the input expressions.

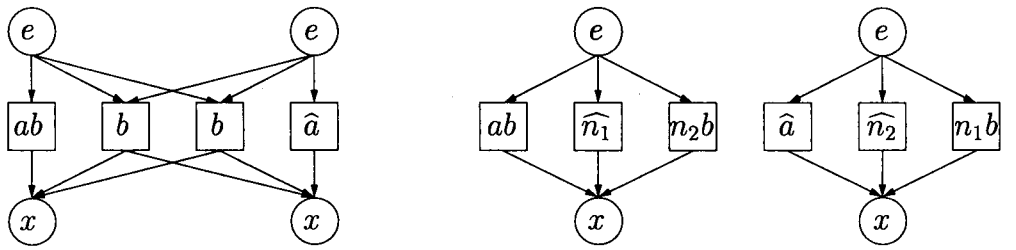
5.3 Synchronisation (Part I)

This section gives an overview of how the axiomatisation for the syntax in Table 4.1 may be extended to duplication equivalence. A similar approach to that taken in Section 5.2 for the basic syntax is used. A rigorous treatment of the problem is not given – rather an outline of a solution is presented.

5.3.1 Background

The synthesis algorithm of Chapter 4 can be extended in the same way as described in Section 5.2 by removing all the duplicate places and transitions from the input net. Propositions 1 and 38 are easily extended to a syntax which includes the synchronisation operator. Therefore, in order to show that this approach to the synthesis algorithm is valid, it remains to show that for any implementation, Σ, of a box expression, E over the syntax in Table 4.1, then the net obtained by removing all duplicate transitions from Σ is the implementation of a Box expression over the syntax in Table 4.3. It has already been shown that duplicate transitions arising from the basic syntax can be removed. For duplicates arising from synchronisation, there is no problem because the scoping form represents each transition independently. In fact the property still holds if it is required that the duplicate free version of Σ is an implementation of an expression from the syntax in Table 4.1. This is due to the fact that synchronisation operations cannot overlap each other – *i.e.* the operations must either operate on disjoint subexpressions, or one synchronisation must be entirely within the scope of the second synchronisation.

When the scoping form is used, there may be several atomic actions relating to a single duplicated transition. For example, in Figure 5.4, net (i) shows the implementation of $(\{a, b\} \parallel \widehat{a})$ **sy** a **sy** a. A scoping form representation for



net(i) *net(ii)*

Figure 5.4: Duplication arising from synchronisation

this expression (*i.e.* in the form that would be produced by the synthesis algorithm given net (i) as input) is:

$$E = [\{n_1, n_2\} : (\{a, b\} \,\square\, \widehat{n_1} \,\square\, \{n_2, b\}) \parallel (\hat{a} \,\square\, \widehat{n_2} \,\square\, \{n_1, b\})]$$

The duplicate transitions in net (i) arise from the two pairs of actions $\widehat{n_1}, \{n_1, b\}$ and $\{n_2, b\}, \widehat{n_2}$ in E. Net (ii) in Figure 5.4 shows the implementation of E before the scoping operation is applied. It can be seen that the representation of the duplicated transitions is distributed throughout the expression, and that each action composing part of the representation of one of the transitions is in an atomic choice context with an action composing part of the representation of the duplicate transition.

5.3.2 Axiomatisation

The axiomatisation of Section 4.6.4 is extended with the axiom for duplicates arising from the basic syntax introduced in Section 5.2, and rewriting rules to capture the notion of "distributed duplication" described above.

$$\alpha \,\square\, \alpha \;=\; \alpha$$

$$[N_1 \cup N_2 : E] \rightarrow [N_1 : E']$$

$$[N_1 : E] \rightarrow [N_1 \cup N_2 : E']$$

The preconditions and procedure for applying the rewriting rule, $[N_1 \cup N_2 : E] \rightarrow [N_1 : E']$ are as follows. Let X be the set of action names in E.

- For each basic action $n \in N_1 \cup N_2$, E must contain an action whose label is $\{\widehat{n}\}$:

$$\forall n \in N_1 \cup N_2, \forall x \in X : \widehat{n} \in \mu(x) \Rightarrow \mu(x) = \{\widehat{n}\}$$

  Furthermore, no other action should have a label containing $\widehat{n}$:

$$\forall n \in N_1 \cup N_2, \forall x \in X : \widehat{n} \in \mu(x) \Rightarrow (\forall x' \in X \widehat{n} \in \mu(x') \Rightarrow x' = x)$$

- E must have an action whose label contains one copy of every basic action in N_1. Similarly, there must be an action in E whose label contains one copy of every basic action in N_2.

$$\text{for } 1 \leq i \leq 2, \exists x \in X : \mu(x) \cap N_i = N_i$$

  Furthermore, there can only be one such action for each N_i – the following precondition must hold for $1 \leq i \leq 2$:

$$|\{x \in X : \mu(x) \cap N_i \neq \emptyset\}| = 1$$

  Let x_1 (x_2) be the action whose label contains all the basic actions in N_1 (N_2). It is a precondition of the rewriting rule that the labels of the transitions obtained by scoping on sets N_1 and N_2 must match:

$$\mu(x_1) - N_1 = \mu(x_2) - N_2$$

- Define X_1 (X_2) to be the set of actions in E corresponding to N_1 (N_2):

$$X_i = \{x \in X \mid \mu(x) \cap (N_i \cup \widehat{N_i}) \neq \emptyset\}$$

  It order for the rewriting rule to be applied, it is necessary that there is a bijective mapping, $\beta : X_1 \rightarrow X_2$ such that for all $x \in X_1$, $x \,\square\, \beta(x)$ appears in E. If all the preconditions have been met, then E' is constructed from E by replacing $x \,\square\, \beta(x)$ by x, for all $x \in X_1$.

The procedure for applying the symmetrical rewriting rule, $[N_1 : E] \rightarrow [N_1 \cup N_2 : E']$ is as follows: Let X be the set of action names in E.

- For each basic action $n \in N_1$, E must contain an action whose label is $\{\widehat{n}\}$:

  $$\forall n \in N_1, \forall x \in X : \widehat{n} \in \mu(x) \Rightarrow \mu(x) = \{\widehat{n}\}$$

  Furthermore, no other action should have a label containing $\widehat{n}$:

  $$\forall n \in N_1, \forall x \in X : \widehat{n} \in \mu(x) \Rightarrow (\forall x' \in X \widehat{n} \in \mu(x') \Rightarrow x' = x)$$

- E must have an action whose label contains one copy of every basic action in N_1.

  $$\exists x \in X : \mu(x) \cap N_1 = N_1$$

  Furthermore, there can only be one such action – the following precondition must hold:

  $$|\{x \in X : \mu(x) \cap N_1 \neq \emptyset\}| = 1$$

  Let x_1 be the action whose label contains all the basic actions in N_1, and N_2 be a set of $|N_1|$ new basic actions that are not already being used. A new action, x_2 is created, such that $\mu(x_2) = N_2 + (\mu(x_1) - N_1)$.

- Define X_1 to be the set of actions in E corresponding to N_1:

  $$X_1 = \{x \in X \mid \mu(x) \cap (N_1 \cup \widehat{N_1}) \neq \emptyset\}$$

  Let X_2 be set of actions comprising x_2, and the set of new actions $X' = x'_1, ..., x'_{|N_2|}$ corresponding to each member of N_2 such that for each $n \in N_2$, $\exists x' \in X : \mu(x') = \{\widehat{n}\}$. Choose any bijective mapping, $\beta : X_1 \rightarrow X_2$ and construct E' from E by replacing each $x \in X_1$ in E by $(x_1 \,\square\, \beta(x_1))$.

5.4 Synchronisation (Part II)

The main result of the investigation in this section is a sound and complete axiomatisation of duplication equivalence for a fragment of recursion-free PBC

containing the synchronisation operator. The important difference to the work in Chapter 4, and Section 5.3 is that the axiomatisation is given purely in terms of the synchronisation operator, and the scoping representation is not used. This approach may give a greater insight into properties of the synchronisation operator.

In Section 5.4.1 a class of Petri nets is defined which are used throughout the rest of this Chapter. Section 5.4.2 discusses the relationship between the operation of synchronisation and duplication equivalence. Section 5.7 transfers the results on duplication equivalence obtained for boxes to the domain of box expressions. Section 5.8 contains the proposed axiomatisation of duplication equivalence. It is followed by the proofs of soundness and completeness of that axiomatisation, presented respectively in Sections 5.9 and 5.10. Finally, some of the issues related to the proposed axiomatisation are briefly discussed.

5.4.1 Labelled nets

A transition t is *simple* if $W(s,t) \leq 1$ and $W(t,s) \leq 1$, for every place s. The net is *T-restricted* if the pre-set and post-set of every transition are non-empty. A *labelled net* is a T-restricted net without isolated places. Figure 5.5 shows a labelled net Σ that corresponds to the box expression $((a\|c) \,\Box\, b); d$.

The different components of the net Σ will often be decorated with the index $_\Sigma$. The same convention will apply to other notations subsequently introduced. The notation $n_1 \ldots n_k \bowtie m_1 \ldots m_l$ means that the 'sum' of the weight functions of nodes $n_1, \ldots, n_k$ is the same as the 'sum' of the weight functions of nodes $m_1, \ldots, m_l$. That is, for every node n in Σ,

$$\sum_{i=1}^{k} W(n_i, n) = \sum_{i=1}^{l} W(m_i, n) \quad \text{and} \quad \sum_{i=1}^{k} W(n, n_i) = \sum_{i=1}^{l} W(n, m_i).$$

In other words, the nodes $n_1, \ldots, n_k$ have the same *connectivity* as $m_1, \ldots, m_l$. Note that for the net of Figure 5.5, $t_1 \bowtie_\Sigma t_0 t_2$. To simplify some of the definitions, δ will be used to denote a 'dummy' simple transition which, if present, would satisfy ${}^\bullet\delta = {}^\bullet\Sigma$ and $\delta^\bullet = \Sigma^\bullet$. For example, $t \bowtie_\Sigma \delta u$ should

be interpreted as signifying that $W(s,t) = W(s,u) + 1$, for all $s \in {}^\bullet\Sigma$, and $W(s,t) = W(s,u)$ otherwise; and that $W(t,s) = W(u,s) + 1$, for all $s \in \Sigma^\bullet$, and $W(t,s) = W(u,s)$ otherwise. The nodes $n_1, \ldots, n_k$ have *constant connectivity* with a set of nodes N if, for all $n, m \in N$,

$$\sum_{i=1}^{k} W(n_i, n) = \sum_{i=1}^{k} W(n_i, m) \quad \text{and} \quad \sum_{i=1}^{k} W(n, n_i) = \sum_{i=1}^{k} W(m, n_i)$$

Constant connectivity will be denoted by $(n_1 \ldots n_k, N) \in \mathsf{const}_\Sigma$. For example, in Figure 5.5, $(t_0 t_2, \{s_0, s_1\}) \in \mathsf{const}_\Sigma$. Directly from the definition of $\bowtie$:

Proposition 40 If $n_1 \bowtie m_1, \ldots, n_k \bowtie m_k$ then $n_1 \ldots n_k \bowtie m_1 \ldots m_k$. Conversely, if $n_1 \ldots n_k \bowtie m_1 \ldots m_k$ and $n_1 \ldots n_{k-1} \bowtie m_1 \ldots m_{k-1}$ then $n_k \bowtie m_k$.
□

$n \simeq m$ is used to indicate that nodes n, m of a net, Σ are duplicates (*i.e.* $n =_{dup} m$). Clearly, $\simeq$ is an equivalence relation; its equivalence class containing node n will be represented by $[n]_\simeq$.

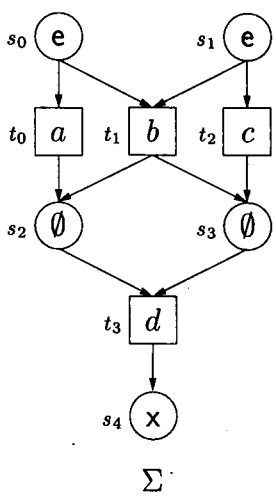


Figure 5.5: A labelled net.

Net union

Net union is a partial operation defined only for pairs of *unionable* nets which means that their transition sets are disjoint and their label functions coincide on the common places. The *union* $\Sigma_1 \cup \Sigma_2$ of two unionable nets, Σ_1 and Σ_2, is defined as a net with the node set being the union of the nodes of Σ_1 and Σ_2, and the weight and label functions being inherited from Σ_1 and Σ_2 (if the value for a weight of the new net cannot be found in the original nets, it is set to zero). Figure 5.6 shows two unionable nets, Σ_1 and Σ_2, and their union $\Sigma_1 \cup \Sigma_2$.

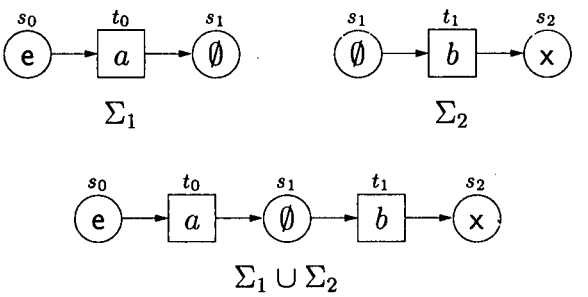


Figure 5.6: Net union.

Net union will usually be applied when the common places can be partitioned into $\otimes$-sets created by the operation of place multiplication. Let Σ_1 and Σ_2 be unionable nets. A non-empty set of places $P \subseteq S_{\Sigma_1} \cap S_{\Sigma_2}$ is a *$\otimes$-set* if for all $s, r \in P$ there is $p \in P$ such that $s \simeq_{\Sigma_1} p$ and $r \simeq_{\Sigma_2} p$.

Proposition 41 Let P be a $\otimes$-set for two unionable nets Σ_1 and Σ_2. If $t_1, \ldots, t_m$ are transitions in Σ_1 and $v_1, \ldots, v_k, \ldots, v_{k+l}$ are transitions in Σ_2 such that $m, l \geq 1$, $k \geq 0$ and

$$t_1 \ldots t_m v_1 \ldots v_k \bowtie_{\Sigma_1 \cup \Sigma_2} v_{k+1} \ldots v_{k+l}$$

then ${}^\bullet t_1 \cup t_1{}^\bullet \cup \ldots \cup {}^\bullet t_m \cup t_m{}^\bullet \subseteq S_{\Sigma_1} \cap S_{\Sigma_2}$ and $(t_1 \ldots t_m, P) \in \mathsf{const}_{\Sigma_1}$.

Proof: The first part follows directly from the definitions of $\bowtie$ and net union. To show $(t_1 \ldots t_m, P) \in \mathsf{const}_{\Sigma_1}$, assume $s, r \in P$. Then, by P being a

$\otimes$-set, there is $p \in P$ such that $s \simeq_{\Sigma_1} p$ and $r \simeq_{\Sigma_2} p$. Denote, for $x \in \{s, r, p\}$,

$$\kappa(x) = \sum_{i=k+1}^{k+l} W_{\Sigma_2}(x, v_i) - \sum_{i=1}^{k} W_{\Sigma_2}(x, v_i) \quad \text{and} \quad \mu(x) = \sum_{i=1}^{m} W_{\Sigma_1}(x, t_i).$$

From $t_1 \ldots t_m v_1 \ldots v_k \bowtie_{\Sigma_1 \cup \Sigma_2} v_{k+1} \ldots v_{k+l}$ it follows that $\mu(x) = \kappa(x)$ for $x \in \{s, r, p\}$. Moreover, from $s \simeq_{\Sigma_1} p$ and $r \simeq_{\Sigma_2} p$ respectively, $\mu(s) = \mu(p)$ and $\kappa(r) = \kappa(p)$ are obtained. Hence

$$\mu(s) = \mu(p) = \kappa(p) = \kappa(r) = \mu(r).$$

This and symmetry of the argument yield $(t_1 \ldots t_m, P) \in \mathsf{const}_{\Sigma_1}$. □

Duplication equivalence

Let Σ be a labelled net. The *duplication quotient* of Σ is defined to be a labelled net

$$[\Sigma]_\simeq = (\{ [s]_\simeq \mid s \in S \}, \{ [t]_\simeq \mid t \in T \}, W', \lambda')$$

where for nodes n and m in Σ, $\lambda(n) = \lambda'([n]_\simeq)$ and $W(n, m) = W'([n]_\simeq, [m]_\simeq)$. Figure 5.7 shows a labelled net Σ and its duplication quotient $[\Sigma]_\simeq$. A notion central to the approach used is now introduced. Labelled nets Σ_1 and Σ_2 are *duplication equivalent* if their duplication quotients are isomorphic nets. This is denoted by $\Sigma_1 \simeq \Sigma_2$ or $\Sigma_1 \simeq_h \Sigma_2$, where h is an isomorphism for $[\Sigma_1]_\simeq$ and $[\Sigma_2]_\simeq$. In the latter case $n \simeq_h m$ may be written if n and m are two nodes of respectively Σ_1 and Σ_2 such that $h([n]_\simeq) = [m]_\simeq$. As it was shown in [5], $\simeq$ is an equivalence relation. This can be slightly strengthened as follows: For all labelled nets Σ_1, Σ_2 and Σ_3,

$$\Sigma_2 \simeq_h \Sigma_2 \text{ and } \Sigma_2 \simeq_g \Sigma_3 \text{ implies } \Sigma_1 \simeq_{h \circ g} \Sigma_3. \tag{5.1}$$

The next result establishes some basic relationships between being a duplicate, having the same connectivity, and being duplication equivalent nets.

Proposition 42 Let $\Sigma_1 \simeq_h \Sigma_2$ be duplication equivalent labelled nets.

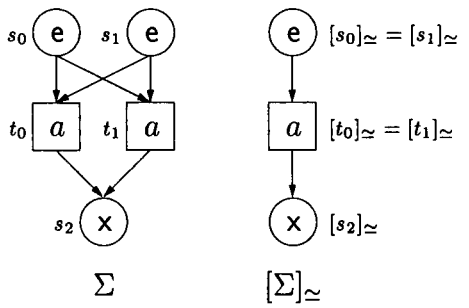


Figure 5.7: Labelled net and its duplication quotient.

1. If $n_1 \simeq_h m_1, \ldots, n_{k+l} \simeq_h m_{k+l}$ $(k, l \geq 1)$ and $n_1 \ldots n_k \bowtie_{\Sigma_1} n_{k+1} \ldots n_{k+l}$ then $m_1 \ldots m_k \bowtie_{\Sigma_2} m_{k+1} \ldots m_{k+l}$.

2. If $n \bowtie_{\Sigma_1} n' \simeq_h m' \bowtie_{\Sigma_2} m$ and $\lambda_{\Sigma_1}(n) = \lambda_{\Sigma_2}(m)$ then $n \simeq_h m$.

3. If $n \simeq_h m \bowtie_{\Sigma_2} m' \simeq_{h^{-1}} n'$ then $n \bowtie_{\Sigma_1} n'$.

Proof: (1) Let m be a node in Σ_2. Then there is a node n in Σ_1 such that $n \simeq_h m$. For every $i \leq k + l$, by $n_i \simeq_h m_i$, $W_{\Sigma_1}(n, n_i) = W_{\Sigma_2}(m, m_i)$. Hence

$$\sum_{i=1}^{k} W_{\Sigma_2}(m, m_i) = \sum_{i=1}^{k} W_{\Sigma_1}(n, n_i) = \sum_{i=k+1}^{k+l} W_{\Sigma_1}(n, n_i) = \sum_{i=k+1}^{k+l} W_{\Sigma_2}(m, m_i).$$

This and symmetry of the argument yield $m_1 \ldots m_k \bowtie_{\Sigma_2} m_{k+1} \ldots m_{k+l}$.

(2) There is q such that $n \simeq_h q$. It suffices to show that $m \bowtie_{\Sigma_2} q$. Take any node r in Σ_2. It is only shown that $W_{\Sigma_2}(m, r) = W_{\Sigma_2}(q, r)$. There is p in Σ_1 such that $p \simeq_h r$. $W_{\Sigma 1}(n, p) = W_{\Sigma 2}(q, r)$ and $W_{\Sigma_1}(n', p) = W_{\Sigma_2}(m', r)$. Moreover, by $n \bowtie_{\Sigma 1} n'$ and $m \bowtie_{\Sigma_2} m'$, $W_{\Sigma_1}(n, p) = W_{\Sigma_1}(n', p)$ and $W_{\Sigma_2}(m, r) = W_{\Sigma_2}(m', r)$. Hence $W_{\Sigma_2}(m, r) = W_{\Sigma_2}(m', r) = W_{\Sigma_1}(n', p) = W_{\Sigma_1}(n, p) = W_{\Sigma_2}(q, r)$.

(3) Suppose that p is a node in Σ_1. Then there is r such that $p \simeq_h r$. By $m \bowtie_{\Sigma_2} m'$, $W_{\Sigma_2}(m, r) = W_{\Sigma_2}(m', r)$. Moreover, by $n \simeq_h m$ and $n' \simeq_h m'$, $W_{\Sigma_1}(n, p) = W_{\Sigma_2}(m, r)$ and $W_{\Sigma_1}(n', p) = W_{\Sigma_2}(m', r)$. Hence $W_{\Sigma_1}(n, p) = W_{\Sigma_1}(n', p)$. Similarly, $W_{\Sigma_1}(p, n) = W_{\Sigma_1}(p, n')$. Thus $n \bowtie_{\Sigma_1} n'$. □

Place-sharing nets

Labelled nets Σ_1 and Σ_2 are known as *place-sharing*, if their place sets and place labellings are exactly the same (no conditions are imposed on the transition sets of the two nets). In such cases, for all transitions, t in Σ_1 and u in Σ_2, $t \simeq_{\Sigma_1\Sigma_2} u$ if $\lambda_{\Sigma_1}(t) = \lambda_{\Sigma_2}(u)$ and for every place s in Σ_1 (and so in Σ_2),

$$W_{\Sigma_1}(t,s) = W_{\Sigma_2}(u,s) \quad \text{and} \quad W_{\Sigma_1}(s,t) = W_{\Sigma_2}(s,u).$$

Intuitively, $t \simeq_{\Sigma_1\Sigma_2} u$ means that t and u are 'remote' duplicates since they have the same connectivity if one looks at the places of the two nets. Moreover, an isomorphism h for the duplication quotients of Σ_1 and Σ_2 establishing duplication equivalence of Σ_1 and Σ_2 will be called *place-preserving* if $s \simeq_h s$, for every place s in the two nets. This is denoted by $\Sigma_1 \cong_h \Sigma_2$ or $\Sigma_1 \cong \Sigma_2$. The following proposition is a fundamental result concerning duplication equivalence of place-sharing nets.

Proposition 43 Let Σ_1 and Σ_2 be place-sharing labelled nets.

1. If $\Sigma_1 \cong_h \Sigma_2$ then for all transitions, t in Σ_1 and u in Σ_2, $t \simeq_h u$ if and only if $t \simeq_{\Sigma_1\Sigma_2} u$.

2. If $T_{\Sigma_1} = \{t \mid t \simeq_{\Sigma_1\Sigma_2} u\}$ and $T_{\Sigma_1} = \{u \mid t \simeq_{\Sigma_1\Sigma_2} u\}$ then $\Sigma_1 \cong \Sigma_2$.

Proof: Denote $S = S_{\Sigma_1} = S_{\Sigma_2}$.

(1) Suppose $t \simeq_h u$. Then $\lambda_{\Sigma_1}(t) = \lambda_{\Sigma_2}(u)$. Let s be any place in S. Since $s \simeq_h s$, $W_{\Sigma_1}(t,s) = W_{\Sigma_2}(u,s)$ and $W_{\Sigma_1}(s,t) = W_{\Sigma_2}(s,u)$. Hence $t \simeq_{\Sigma_1\Sigma_2} u$.

Suppose $t \simeq_{\Sigma_1\Sigma_2} u$. Then $\lambda_{\Sigma_1}(t) = \lambda_{\Sigma_2}(u)$. Let $t \simeq_h w$ and s be a place in S. Therefore $W_{\Sigma_2}(u,s) = W_{\Sigma_1}(t,s) = W_{\Sigma_2}(w,s)$ and $W_{\Sigma_2}(s,u) = W_{\Sigma_1}(s,t) = W_{\Sigma_2}(s,w)$. Hence $u \simeq_{\Sigma_2} w$ and so $t \simeq_h u$.

(2) Define a mapping from the nodes of the duplication quotient of Σ_1 to the nodes of the duplication quotient of Σ_2 thus:

$$h = \{([s]_{\simeq}^{\Sigma_1}, [s]_{\simeq}^{\Sigma_2}) \mid s \in S\} \cup \{([t]_{\simeq}^{\Sigma_1}, [u]_{\simeq}^{\Sigma_2}) \mid t \simeq_{\Sigma_1\Sigma_2} u\}.$$

One can see that $\Sigma_1 \cong_h \Sigma_2$. To show that h is a bijection, it suffices to show that: (i) if $s, r \in S$ and $s \simeq_{\Sigma_1} r$ then $s \simeq_{\Sigma_2} r$; and (ii) if t, u, t', u' are transitions such that $u \simeq_{\Sigma_2\Sigma_1} t \simeq_{\Sigma_1} t' \simeq_{\Sigma_1\Sigma_2} u'$ then $u \simeq_{\Sigma_2} u'$. To show (i), take any $y \in T_{\Sigma_2}$. Then, by the assumption made in 43(2), there is $z \in T_{\Sigma_1}$ such that $z \simeq_{\Sigma_1\Sigma_2} y$. Therefore:

$$W_{\Sigma_2}(y, s) = W_{\Sigma_1}(z, s) = W_{\Sigma_1}(z, r) = W_{\Sigma_2}(y, r).$$

The proof that $W_{\Sigma_2}(s, y) = W_{\Sigma_2}(r, y)$ is similar. To show (ii), take any $s \in S$. Then

$$W_{\Sigma_2}(s, u) = W_{\Sigma_1}(s, t) = W_{\Sigma_1}(s, t') = W_{\Sigma_2}(s, u').$$

The proof that $W_{\Sigma_2}(u, s) = W_{\Sigma_2}(u', s)$ is similar.

What remains to be shown is that if $s \in S$ and $t \simeq_{\Sigma_1\Sigma_2} u$ then the weight of the arc between $[s]_{\simeq}^{\Sigma_1}$ and $[t]_{\simeq}^{\Sigma_1}$ in $[\Sigma_1]_{\simeq}$ is the same as the weight of the arc between $[s]_{\simeq}^{\Sigma_2}$ and $[u]_{\simeq}^{\Sigma_2}$ in $[\Sigma_2]_{\simeq}$. This, however, follows directly from $W_{\Sigma_1}(s, t) = W_{\Sigma_2}(s, u)$. □

Note that from the first part of the proposition it follows that there can be at most one place-preserving isomorphism (between the respective duplication quotients) establishing duplication equivalence of two place-sharing nets.

5.4.2 Synchronisation

A *synchronisation set* is a set of communication actions A which contains the conjugates of all its actions, i.e. $\widehat{A} = A$. For every communication action a, a denotes the synchronisation set $\{a, \widehat{a}\}$. As in CCS, it is implicitly assumed that two transitions labelled with conjugate communication actions can be synchronised to yield a new transition labelled with the internal action.[1] Two transitions, t and u, whose labels are conjugates belonging to a synchronisation set A are called *A–synchronisable*, $(t, u) \in \mathsf{syn}_A$.

[1] The synchronisation mechanism used in this section is basically that of CCS, since multi actions are not used

The *synchronisation* of a labelled net Σ by a synchronisation set A is a net Σ **sy** A which is defined as Σ extended by a set of new transitions. Exactly one new transition, $t \diamond u$, is added for every pair of A–synchronisable transitions of Σ, t and u. The label of $t \diamond u$ is $\emptyset$ and the weight function is extended so that $t \diamond u \bowtie_\Sigma \mathbf{sy}_A\ tu$. It is assumed that $t \diamond u$ is the same as $u \diamond t$. Figure 5.8 shows two consecutive applications of the synchronisation operator. Note that Σ **sy** a and (Σ **sy** a) **sy** a are duplication equivalent, but not isomorphic. Thus synchronisation is not idempotent with respect to net isomorphism. However, it is idempotent with respect to duplication equivalence which was one of the reasons for introducing duplication equivalence in [5].

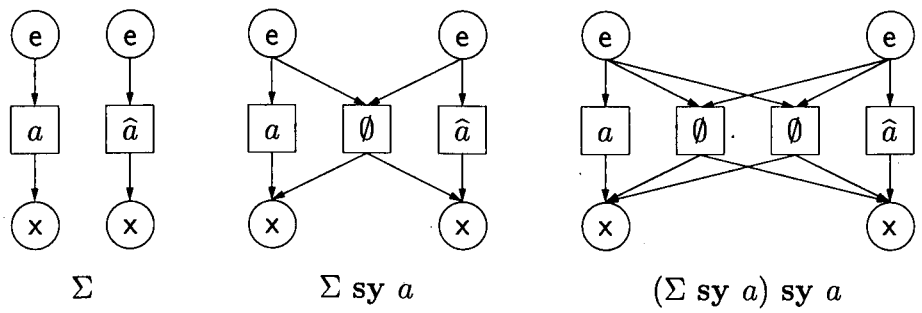


Figure 5.8: Synchronisation (place labels omitted).

Having the same connectivity is preserved through synchronisation.

Proposition 44 Let Σ be a labelled net with nodes $n_1, \ldots, n_k, m_1, \ldots, m_l$ and A be a synchronisation set. Then

$$n_1 \ldots n_k \bowtie_\Sigma m_1 \ldots m_l \quad \text{if and only if} \quad n_1 \ldots n_k \bowtie_\Sigma \mathbf{sy}_A\ m_1 \ldots m_l.$$

Proof: That ($\Leftarrow$) implication holds follows directly from the definition of synchronisation. Denote $\Sigma' = \Sigma$ **sy** A. To show the reverse implication, suppose $t = u \diamond w \in T_{\Sigma'} - T_\Sigma$. By $n_1 \ldots n_k \bowtie_\Sigma m_1, \ldots, m_l$,

$$\sum_{i=1}^{k} W_\Sigma(n_i, u) = \sum_{i=1}^{l} W_\Sigma(m_i, u) \quad \text{and} \quad \sum_{i=1}^{k} W_\Sigma(n_i, w) = \sum_{i=1}^{l} W_\Sigma(m_i, w).$$

Hence,

$$\begin{aligned}
\sum_{i=1}^{k} W_{\Sigma'}(n_i, t) &= \sum_{i=1}^{k} (W_{\Sigma'}(n_i, u) + W_{\Sigma'}(n_i, w)) \\
&= \sum_{i=1}^{k} W_{\Sigma}(n_i, u) + \sum_{i=1}^{k} W_{\Sigma}(n_i, w) \\
&= \sum_{i=1}^{l} W_{\Sigma}(m_i, u) + \sum_{i=1}^{l} W_{\Sigma}(m_i, w) \\
&= \sum_{i=1}^{l} (W_{\Sigma'}(m_i, u) + W_{\Sigma'}(m_i, w)) \\
&= \sum_{i=1}^{l} W_{\Sigma'}(m_i, t).
\end{aligned}$$

Similarly, one may show that $\sum_{i=1}^{k} W_{\Sigma'}(t, n_i) = \sum_{i=1}^{l} W_{\Sigma'}(t, m_i)$. □

Duplication equivalence is preserved through synchronisation. What is more, the isomorphism establishing the equivalence can also be preserved.

Proposition 45 Let $\Sigma_1 \simeq_h \Sigma_2$ be duplication equivalent labelled nets and A be a synchronisation set. Then there is an isomorphism g such that Σ_1 **sy** $A \simeq_g \Sigma_2$ **sy** A and, for all nodes n and m in respectively Σ_1 and Σ_2, $n \simeq_h m$ if and only if $n \simeq_g m$.

Proof: Denote $\Sigma_a = \Sigma_1$ **sy** A and $\Sigma_b = \Sigma_2$ **sy** A. Define

$$\begin{aligned}
g \;=\; & \{\, ([n]_{\simeq}^{\Sigma_a}, [m]_{\simeq}^{\Sigma_b}) \mid n \simeq_h m \,\} \;\cup\; \{\, ([t \diamond u]_{\simeq}^{\Sigma_a}, [v \diamond w]_{\simeq}^{\Sigma_b}) \\
& \mid (t, u) \in \mathsf{syn}_A \wedge t \simeq_h v \wedge u \simeq_h w \,\}.
\end{aligned}$$

A straightforward yet rather lengthy argument can show that g is a required isomorphism. □

Σ_1 and Σ_1 **sy** A are place-sharing nets. Hence, directly from proposition 43, a necessary and sufficient condition for the existence of a place-preserving isomorphism for the duplication quotients of Σ_1 and Σ_1 **sy** A is obtained.

Corollary 7 Let Σ be a labelled net and A be a synchronisation set. Then $\Sigma \cong \Sigma$ **sy** A if and only if for every transition t in Σ **sy** A there is u in Σ such that $t \simeq_{\Sigma \textbf{ sy } A} u$.

Proof: The ($\Rightarrow$) implication follows from proposition 43(1), and the ($\Leftarrow$) implication from proposition 43(2). □

The next result can be interpreted as saying that being duplication equivalent to a synchronised net makes the synchronisation implicit.

Proposition 46 Let Σ_1 and Σ_2 be labelled nets and A be a synchronisation set such that $\Sigma_1 \simeq \Sigma_2$ **sy** A. Then $\Sigma_1 \cong \Sigma_1$ **sy** A.

Proof: Let $\Sigma_a = \Sigma_1$ **sy** A and $\Sigma_b = \Sigma_2$ **sy** A. By corollary 7, it suffices to show that for every transition $u \diamond w \in T_{\Sigma_a} - T_{\Sigma_1}$, there is $t \in T_{\Sigma_1}$ such that $t \simeq_{\Sigma_a} u \diamond w$. Suppose that $\Sigma_1 \simeq_g \Sigma_b$. There are v and z such that $u \simeq_g v$ and $w \simeq_g z$. Clearly, $v \diamond z \in T_{\Sigma_b}$. Hence there is $t \in T_{\Sigma_1}$ such that $t \simeq_g v \diamond z$. By proposition 42(1) and $v \diamond z \bowtie_{\Sigma_b} vz$, $t \bowtie_{\Sigma_1} uw$ which in turn means that $t \simeq_{\Sigma_a} u \diamond w$. □

The next proposition gathers together a number of simple facts involving synchronisation and duplication equivalence. Note that the second item implies that as far as duplication equivalence is concerned, synchronisation is idempotent and commutative.

Proposition 47 Let Σ be a labelled net, and A and B be synchronisation sets.

1. $\Sigma \simeq \Sigma$ **sy** A if and only if $\Sigma \cong \Sigma$ **sy** A.
2. Σ **sy** A **sy** $B \cong \Sigma$ **sy** $(A \cup B)$.
3. If Σ **sy** $A \simeq \Sigma$ **sy** B then Σ **sy** $A \cong \Sigma$ **sy** B.
4. If $A \subseteq B$ and $\Sigma \simeq \Sigma$ **sy** B then $\Sigma \cong \Sigma$ **sy** A.
5. If $\Sigma \simeq \Sigma$ **sy** A and $\Sigma \simeq \Sigma$ **sy** B then $\Sigma \cong \Sigma$ **sy** $(A \cup B)$.

Proof: (1) Follows directly from proposition 46 and the definition of $\cong$.

(2) Follows directly from proposition 43(2) and the definition of synchronisation.

(3) By Σ **sy** $A \simeq \Sigma$ **sy** B and proposition 46, there are g and f such that

$$\Sigma \textbf{ sy } A \cong_g \Sigma \textbf{ sy } A \textbf{ sy } B \quad \text{and} \quad \Sigma \textbf{ sy } B \cong_f \Sigma \textbf{ sy } B \textbf{ sy } A.$$

Moreover, by proposition 47(2), there are d and e such that

$$\Sigma \textbf{ sy } A \textbf{ sy } B \cong_d \Sigma \textbf{ sy } (A \cup B) \quad \text{and} \quad \Sigma \textbf{ sy } B \textbf{ sy } A \cong_e \Sigma \textbf{ sy } (B \cup A).$$

Hence, by property (5.1) and $A \cup B = B \cup A$, $h = g \circ d \circ e^{-1} \circ f^{-1}$ is an isomorphism satisfying Σ **sy** $A \cong_h \Sigma$ **sy** B.

(4) Follows from Σ **sy** $B = \Sigma$ **sy** $(B-A)$ **sy** A and proposition 46.

(5) From propositions 47(2) and 45, $\Sigma \simeq \Sigma$ **sy** $A \simeq \Sigma$ **sy** B **sy** $A \simeq \Sigma$ **sy** $(A \cup B)$. Then proposition 47(1) may be applied. □

Finally, it is shown that being a $\otimes$-set is preserved through synchronisation.

Proposition 48 If P is a $\otimes$-set for unionable labelled nets Σ_1 and Σ_2, and A, B are synchronisation sets then P is also a $\otimes$-set for Σ_1 **sy** A and Σ_2 **sy** B.

Proof: Follows directly from the definition of $\otimes$-sets and proposition 44. □

If one looks at the last item of proposition 47 then it is obvious that for every net Σ_1 there exists the *maximal* [2] *synchronisation set* A such that $\Sigma_1 \simeq \Sigma_1$ **sy** A. This set will be denoted by max_{Σ_1}. Note that

$$\mathsf{max}_{\Sigma_1} = \bigcup \{A \mid \Sigma_1 \simeq \Sigma_1 \textbf{ sy } A\}.$$

Hence, by proposition 46, if $\Sigma_1 \simeq \Sigma_2$ then $\mathsf{max}_{\Sigma_1} = \mathsf{max}_{\Sigma_2}$.

Net union, duplication equivalence and synchronisation

Crucial to the further development is the way in which combining the net union and synchronisation operations affect duplication equivalence. Firstly, it is

[2] Maximal w.r.t. set inclusion.

shown that from the point of view of duplication equivalence, synchronisation propagates through net union.

Proposition 49 Let $\Sigma_1, \ldots, \Sigma_k$ be pairwise unionable labelled nets and A be a synchronisation set. Then

$$(\Sigma_1 \cup \ldots \cup \Sigma_k) \textbf{ sy } A \;\cong\; ((\Sigma_1 \textbf{ sy } A) \cup \ldots \cup (\Sigma_k \textbf{ sy } A)) \textbf{ sy } A.$$

Proof: Let $\Sigma = (\Sigma_1 \cup \ldots \cup \Sigma_k)$ **sy** A and $\Sigma' = ((\Sigma_1$ **sy** $A) \cup \ldots \cup (\Sigma_k$ **sy** $A))$ **sy** A. Since Σ and Σ' satisfy the condition in proposition 43(2), $\Sigma \cong \Sigma'$. □

A key result on the relationship between net union, synchronisation and duplication equivalence is now formulated.

Proposition 50 Let $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_k$ where $\Sigma_1, \ldots, \Sigma_k$ are pairwise unionable nets, and let A and B be synchronisation sets such that for all distinct $i, j \leq k$, the following hold.

- Σ_i **sy** $A \simeq \Sigma_i$ **sy** B.
- For all $t \in T_i$ and $v \in T_j$, if $(t, v) \in \mathsf{syn}_A$ (or $(t, v) \in \mathsf{syn}_B$) then there is $w \in T_\Sigma$ such that $\lambda_\Sigma(w) = \emptyset$ and $tv \bowtie_\Sigma w$, or there are $u, w \in T_\Sigma$ such that $(u, w) \in \mathsf{syn}_B$ (resp. $(u, w) \in \mathsf{syn}_A$) and $tv \bowtie_\Sigma uw$.

Then Σ **sy** $A \cong \Sigma$ **sy** B.

Proof: Let $\Sigma_a = \Sigma$ **sy** A and $\Sigma_b = \Sigma$ **sy** B. From propositions 47(3) and 43(1) it follows that proposition 43(2) may be applied. Hence $\Sigma_a \cong \Sigma_b$. □

Later, a specific instance of the last result will be used for $B = A \cup \mathsf{a}$.

Corollary 8 Let $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_k$ where $\Sigma_1, \ldots, \Sigma_k$ are pairwise unionable nets, and let A be a synchronisation set and $a \notin A$ be a communication action such that for all distinct $i, j \leq k$, the following hold.

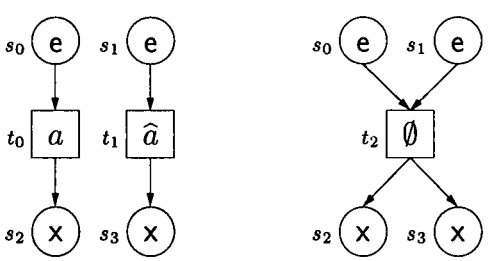
- Σ_i **sy** $(A \cup \mathsf{a}) \simeq \Sigma_i$ **sy** A.

- For all $(t, v) \in (T_i \times T_j) \cap \mathsf{syn_a}$ there is $w \in T_\Sigma$ such that $\lambda_\Sigma(w) = \emptyset$ and $tv \bowtie_\Sigma w$, or there are $u, w \in T_\Sigma$ such that $(u, w) \in \mathsf{syn}_A$ and $tv \bowtie_\Sigma uw$.

Then Σ **sy** $(A \cup \mathsf{a}) \cong \Sigma$ **sy** A. □

Reversing Corollary 8

The aim is to reverse corollary 8 for $k = 2$. This result is required to characterise maximal synchronisation sets of nets which are obtained through composition. In particular, it needs to be shown that if $(\Sigma_1 \cup \Sigma_2)$ **sy** $(A \cup \mathsf{a}) \simeq (\Sigma_1 \cup \Sigma_2)$ **sy** A then Σ_1 **sy** $(A \cup \mathsf{a}) \simeq \Sigma_1$ **sy** A. However, such a result does not, in general, hold. For example, consider the following two nets (note that the union of the two nets in this example corresponds to the box expression $(a \| \widehat{a}) \,\square\, \emptyset$).



Then, clearly, $(\Sigma_1 \cup \Sigma_2)$ **sy** $\mathsf{a} \simeq \Sigma_1 \cup \Sigma_2$ but Σ_1 **sy** $\mathsf{a} \not\simeq \Sigma_1$. Intuitively, the reason why there is a problem here is that, in $\Sigma_1 \cup \Sigma_2$, the transition t_2 can be interpreted both as coming from Σ_2, and as coming from Σ_1 (e.g. by being the 'result' of synchronising transitions t_0 and t_1). Therefore, some constraints are placed on the two nets, Σ_1 and Σ_2, before trying to reverse the corollary. To this end, suppose that Σ_1 and Σ_2 are two unionable labelled nets, A is a synchronisation set, $a \notin A$ is a communication action, and the six assumptions, A1-A6 below also hold.

A1 In Σ_1 and Σ_2, all the transitions labelled with communication actions are simple, and every $\emptyset$-labelled transition t satisfies $W(s,t) \leq 2$ and $W(t,s) \leq 2$, for every place s.

A2 There exist two disjoint non-empty sets of places, E and X, of $Sigma_1$ such that for every transition t in Σ_1, $t^\bullet \cap E = {}^\bullet t \cap X = \emptyset$.

A3 If t is a transition in Σ_1 labelled with a communication action then ${}^\bullet t \cap E = \emptyset \vee {}^\bullet t \subseteq E$ and $t^\bullet \cap X = \emptyset \vee t^\bullet \subseteq X$.

A4 The set of common places of Σ_1 and Σ_2 is either $\emptyset$ or E or X or $E \cup X$.

A5 If E (or X) is included in $S_{\Sigma_1} \cap S_{\Sigma_2}$ then E (resp. X) is a $\otimes$-set for Σ_1 and Σ_2.

A transition t in $(\Sigma_1 \cup \Sigma_2)$ **sy** $(A \cup \mathsf{a})$ is called an *EX-transition*, and denoted $t \in T_{EX}$, if ${}^\bullet t = E$, $t^\bullet = X$ and $(t,E), (t,X) \in \mathsf{const}_{(\Sigma_1 \cup \Sigma_2)}$ **sy** $_{(A \cup \mathsf{a})}$.

A6 If t is an EX-transition in $(\Sigma_1 \cup \Sigma_2)$ **sy** A whose label belongs to $A \cup \{\emptyset\}$, then there is $u \in T_{\Sigma_1}$ **sy** $_A$ such that $t \simeq_{(\Sigma_1 \cup \Sigma_2)}$ **sy** $_A$ u.

Intuitively, E and/or X form the interface between Σ_1 and Σ_2, E being derived from the entry, and X being derived from the exit places of Σ_1 (later, the sets E and X will be derived through the place multiplication and addition operators, which will, in particular, guarantee that A5 holds). In the last example, $E = \{s_0, s_1\}$ and $X = \{s_2, s_3\}$. Note also that this example fails to satisfy A6 for $t = t_2$. It is worth mentioning that A1-A5 are conditions which will be satisfied by all the nets associated with box expressions or being used in the definitions of the composition operators (see proposition 53). However, the nature of condition A6 is different, as the last example has demonstrated. The next result can be seen a reverse of the first part of corollary 8.

Proposition 51 If $(\Sigma_1 \cup \Sigma_2)$ **sy** $(A \cup \mathsf{a}) \simeq (\Sigma_1 \cup \Sigma_2)$ **sy** A then Σ_1 **sy** $(A \cup \mathsf{a}) \simeq \Sigma_1$ **sy** A.

Proof: By corollary 7 and proposition 47(2), it suffices to show that if $t \in T_{\Sigma_1}$ **sy** $_{(A \cup \mathsf{a})} - T_{\Sigma_1}$ **sy** $_A$ then $t \simeq_{\Sigma_1}$ **sy** $_{(A \cup \mathsf{a})}\ w$ for some transition w in Σ_1 **sy** A. From $(\Sigma_1 \cup \Sigma_2)$ **sy** $(A \cup \mathsf{a}) \simeq (\Sigma_1 \cup \Sigma_2)$ **sy** A, corollary 7 and proposition 47(2), it follows that there is u in $(\Sigma_1 \cup \Sigma_2)$ **sy** A such that $t \simeq_{(\Sigma_1 \cup \Sigma_2)}$ **sy** $_{(A \cup \mathsf{a})}\ u$. If $u \in T_{\Sigma_1}$ **sy** $_A$ then $w = u$ may be used. If $u \in T_{\Sigma_2}$ **sy** $_A$ then, from propositions 41 and 48, A2, A4 and A5, it follows that $u \in T_{EX}$. Hence, by A6, there is $w \in T_{\Sigma_1}$ **sy** $_A$ such that $w \simeq_{(\Sigma_1 \cup \Sigma_2)}$ **sy** $_A\ u$. Thus $t \simeq_{\Sigma_1}$ **sy** $_{(A \cup \mathsf{a})}\ w$. Finally, assume that $u = v \diamond z$ where $v \in T_{\Sigma_1}$ and $z \in T_{\Sigma_2}$. Then, again from propositions 41 and 48, A2, A4 and A5, it follows that $z \in T_{EX}$. Hence, by A6 (note that $a \notin A$), there is $y \in T_{\Sigma_1}$ such that $z \simeq_{(\Sigma_1 \cup \Sigma_2)}$ **sy** $_A\ y$. Thus $w = v \diamond y$, which belongs to Σ_1 **sy** A, satisfies $t \simeq_{\Sigma_1}$ **sy** $_{(A \cup \mathsf{a})}\ w$. □

The second part of corollary 8, may now be partially reversed.

Proposition 52 Let $(\Sigma_1 \cup \Sigma_2)$ **sy** $(A \cup \mathsf{a}) \simeq (\Sigma_1 \cup \Sigma_2)$ **sy** A and $(t, v) \in (T_{\Sigma_1} \times (T_{\Sigma_2} - T_{EX})) \cap \mathsf{syn}_{\mathsf{a}}$. Moreover, whenever $t \in T_{EX}$, there is no $x \in T_{\Sigma_2}$ **sy** $_A$ such that $t \diamond v \simeq_{(\Sigma_1 \cup \Sigma_2)}$ **sy** $_{(A \cup \mathsf{a})}\ x$. Then there are transitions $(u, w) \in (T_{\Sigma_1} \times T_{\Sigma_2}) \cap \mathsf{syn}_A$ such that $t \bowtie_{\Sigma_1} u$ and $v \bowtie_{\Sigma_2} w$.

Proof: By corollary 7 and proposition 47(2), there is an $x \in T_{(\Sigma_1 \cup \Sigma_2)}$ **sy** $_A$ such that $t \diamond v \simeq_{(\Sigma_1 \cup \Sigma_2)}$ **sy** $_{(A \cup \mathsf{a})}\ x$. If $x \in T_{\Sigma_2}$ **sy** $_A$ (or $x \in T_{\Sigma_1}$ **sy** $_A$) then, from propositions 41 and 48, A1, A2, A4 and A5, it follows that $t \in T_{EX}$ (resp. $v \in T_{EX}$), a contradiction with the assumptions about t (resp. v) that were made. Hence $x = u \diamond w$ where $u \in T_{\Sigma_1}$ and $w \in T_{\Sigma_2}$. By proposition 43(1), $tv \bowtie_{\Sigma_1 \cup \Sigma_2} uw$. Hence it is required to show that $t \bowtie_{\Sigma_1} u$ and $v \bowtie_{\Sigma_2} w$. By proposition 40, it suffices to show the latter.

Suppose $v \bowtie_{\Sigma_2} w$ does not hold. Without loss of generality, and by A1, it may be assumed that $W_{\Sigma_2}(s, v) = 1$ and $W_{\Sigma_2}(s, w) = 0$, for some $s \in S_{\Sigma_2}$. If $s \notin S_{\Sigma_1} \cap S_{\Sigma_2}$ then a contradiction is obtained with $tv \bowtie_{\Sigma_1 \cup \Sigma_2} uw$. If $s \in S_{\Sigma_1} \cap S_{\Sigma_2}$ then, by $tv \bowtie_{\Sigma_1 \cup \Sigma_2} uw$ and A1, $W_{\Sigma_1}(s, t) = 0$ and $W_{\Sigma_1}(s, u) = 1$. By $s \in {}^\bullet v$ and A2, $s \in E$. If ${}^\bullet t \cap E = \emptyset$ then, by A2, there is $q \in S_{\Sigma_1} - (E \cup X)$

such that $W_{\Sigma_1}(q,t) = 1$. On the other hand, by A3 and $s \in {}^\bullet u$, $W_{\Sigma_1}(q,u) = 0$, contradicting $tv \bowtie_{\Sigma_1 \cup \Sigma_2} uw$. Hence ${}^\bullet t \subseteq E$. As a result, there is $r \in E$ such that $W_{\Sigma_1}(r,t) = 1$. By A5, there is $p \in E$ such that $r \simeq_{\Sigma_1} p$ and $s \simeq_{\Sigma_2} p$. Therefore

$$\begin{array}{rrcl} & W_{\Sigma_1}(p,t) + W_{\Sigma_2}(p,v) & = & 1+1 \\ \text{and} & W_{\Sigma_1}(p,u) + W_{\Sigma_2}(p,w) & \leq & 1+0 \end{array}$$

contradicting $tv \bowtie_{\Sigma_1 \cup \Sigma_2} uw$. □

5.5 Composition operators

In this section, the semantics of the choice, parallel, sequence and iteration operators are given in terms of the composition of place sharing nets. It is a relatively simple exercise to check that the semantics presented here are consistent with the earlier semantics given in Chapter 1. The definitions of the four operators are preceded by three auxiliary notions, viz. place addition, place multiplication and gluing of nets.

To begin with, a mechanism is formalised by which a place may be replaced by a set of other places which inherit its connectivity. Let Σ be a labelled net and $s_1, \ldots, s_k$ be its places. Moreover, let $S_1, \ldots, S_k$ be disjoint non-empty sets of places not in Σ and $l_1, \ldots, l_k \in \{\mathsf{e}, \emptyset, \mathsf{x}\}$. Then

$$\Sigma \oplus \{(s_1, S_1, l_1), \ldots, (s_k, S_k, l_k)\} = (S', T', W', \lambda')$$

is a net such that $S' = S - \{s_1, \ldots, s_k\} \cup (S_1 \cup \ldots \cup S_k)$, $T' = T$ and, for all $n, m \in S' \cup T'$,

$$W'(n,m) = \begin{cases} W(n,m) & \text{if } n,m \in S \cup T \\ W(s_i,m) & \text{if } n \in S_i,\ m \in S \cup T \\ W(n,s_j) & \text{if } n \in S \cup T,\ m \in S_j \\ 0 & \text{if } n \in S_i,\ m \in S_j \end{cases}$$

$$\lambda'(n) = \begin{cases} \lambda(n) & \text{if } n \in S \cup T \\ l_i & \text{if } n \in S_i. \end{cases}$$

Let $\Sigma_1, \ldots, \Sigma_k$ be disjoint labelled nets, and Δ be a *gluing set.* The latter is defined by:

$$\Delta \;=\; \{(S_1^1, \ldots, S_{r_i}^1, l_1), \ldots (S_1^m, \ldots, S_{r_m}^1, l_m)\}$$

where $m, r_1, \ldots, r_m \geq 1$ and $l_1, \ldots, l_m \in \{\mathsf{e}, \emptyset, \mathsf{x}\}$ and each S_i^j is the set of entry places or the set of exit places of one of the nets $\Sigma_1, \ldots, \Sigma_k$. It is assumed that, for every $j \leq k$, both ${}^\bullet\Sigma_j$ and $\Sigma_j{}^\bullet$ can appear in Δ at most once and never in the same element of Δ. With these assumptions, $\Sigma_j : \Delta$ for $j \leq k$, is defined by:

$$\Sigma_j : \Delta \;=\; \Sigma_j \oplus \{(s, [S_1^i \otimes \ldots \otimes S_{r_i}^i]_s, l_i) \mid s \in {}^\bullet\Sigma_j \cup \Sigma_j{}^\bullet \wedge \;\; s \in S_1^i \cup \ldots \cup S_{r_i}^i\}$$

where $[S_1^i \otimes \ldots \otimes S_{r_i}^i]_s = \{p \in S_1^i \otimes \ldots \otimes S_{r_i}^i \mid s \in p\}$. Then the net

$$(\Sigma_1, \ldots, \Sigma_k) : \Delta \;=\; (\Sigma_1 : \Delta) \cup \ldots \cup (\Sigma_k : \Delta)$$

where $\cup$ denotes net union, is a *glued net* obtained from nets $\Sigma_1, \ldots, \Sigma_k$ using the gluing set Δ.

The composition operators are defined for a class of nets called pre-boxes. A *pre-box* is a labelled net Σ such that the following hold:

1. $\Sigma^\bullet \neq \emptyset \neq {}^\bullet\Sigma$.
2. $(\Sigma^\bullet)^\bullet = {}^\bullet({}^\bullet\Sigma) = \emptyset$.
3. All the transitions labelled with communication actions are simple, and every $\emptyset$-labelled transition t satisfies $W(s,t) \leq 2$ and $W(t,s) \leq 2$, for every place s.

Let Σ_1, Σ_2 and Σ_3 be disjoint pre-boxes. The four composition operators are defined thus.

- Sequential $\quad \Sigma_1; \Sigma_2 = (\Sigma_1, \Sigma_2) : \{({\Sigma_1}^\bullet, {}^\bullet\Sigma_2, \emptyset)\}$.
- Choice $\quad \Sigma_1 \,\square\, \Sigma_2 = (\Sigma_1, \Sigma_2) : \{({}^\bullet\Sigma_1, {}^\bullet\Sigma_2, \mathsf{e}), ({\Sigma_1}^\bullet, {\Sigma_2}^\bullet, \mathsf{x})\}$.

- Concurrent $\quad \Sigma_1 \| \Sigma_2 = \Sigma_1 \cup \Sigma_2.$

- Iteration $\quad [\Sigma_1 * \Sigma_2 * \Sigma_3] = (\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_1', \Sigma_2', \Sigma_3') : \Delta$

where in the last case Σ_i' is a disjoint copy of Σ_i, for $i = 1, 2, 3$, and Δ is a gluing set given by

$$\begin{aligned} \Delta \;=\; & \{({}^\bullet\Sigma_1, {}^\bullet\Sigma_1', \mathsf{e}), (\Sigma_1{}^\bullet, {}^\bullet\Sigma_2, \Sigma_2'{}^\bullet, {}^\bullet\Sigma_3, \emptyset), \\ & (\Sigma_1'{}^\bullet, \Sigma_2{}^\bullet, {}^\bullet\Sigma_2', {}^\bullet\Sigma_3', \emptyset), (\Sigma_3{}^\bullet, \Sigma_3'{}^\bullet, \mathsf{x})\}. \end{aligned}$$

The sets of places resulting from place multiplication in the definition of the three composition operators can easily be seen as $\otimes$-sets of the composed nets. Consider, for example, the sequential composition

$$\Sigma_1; \Sigma_2 \;=\; \Sigma_a; \Sigma_b \;=\; \Sigma_1 : \{(\Sigma_1{}^\bullet, {}^\bullet\Sigma_2, \emptyset)\} \cup \Sigma_2 : \{(\Sigma_1{}^\bullet, {}^\bullet\Sigma_2, \emptyset)\}.$$

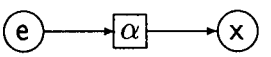
Then $\Sigma_1{}^\bullet \otimes {}^\bullet\Sigma_2$ is a $\otimes$-set for the nets Σ_a and Σ_b. Indeed, $s, r \in \Sigma_1{}^\bullet \otimes {}^\bullet\Sigma_2$ is taken then $s = \{s_1, s_2\}$ and $r = \{r_1, r_2\}$, where $s_1, r_1 \in \Sigma_1{}^\bullet$ and $s_2, r_2 \in {}^\bullet\Sigma_2$. Then $p = \{s_1, r_2\}$ also belongs to $\Sigma_1{}^\bullet \otimes {}^\bullet\Sigma_2$ and satisfies $s \simeq_{\Sigma_a} p$ and $r \simeq_{\Sigma_b} p$.

5.6 Boxes

A class of process expressions is now considered. In this section a subset of the Petri Box Calculus [5] described by the following syntax of *box expressions*:

$$E \;:=\; \alpha \mid E \;\mathbf{sy}\; A \mid E; E \mid E \,\square\, E \mid E \| E \mid [E * E * E] \qquad (5.2)$$

In the above, α is an action in $\mathcal{A} \cup \{\emptyset\}$ and A is a synchronisation set. The five operators correspond to those introduced for labelled nets. There is a mapping which associates with every box expression, E, a labelled net, $\mathsf{box}(E)$, in the following way:

$$
\begin{aligned}
\mathsf{box}(\alpha) &= \text{  } \\
\mathsf{box}(E \text{ sy } A) &= \mathsf{box}(E) \text{ sy } A \\
\mathsf{box}(E; F) &= \mathsf{box}(E); \mathsf{box}(F) \\
\mathsf{box}(E \,[]\, F) &= \mathsf{box}(E) \,[]\, \mathsf{box}(F) \\
\mathsf{box}(E \| F) &= \mathsf{box}(E) \| \mathsf{box}(F) \\
\mathsf{box}([E * F * G]) &= [\mathsf{box}(E) * \mathsf{box}(F) * \mathsf{box}(G)].
\end{aligned}
$$

In what follows, a *box* is a net which can be derived from a box expression through the box() mapping. In general, isomorphic boxes will be identified. The following collection of simple facts about boxes can easily be proven by induction on the structure of expressions generating them.

Proposition 53 Let Σ be a box, t be its transition and s be its place.

1. Σ is a pre-box (and so also a labelled net).

2. If t is labelled with a communication action then

   (a) ${}^\bullet t \cap {}^\bullet\Sigma \neq \emptyset$ implies ${}^\bullet t \subseteq {}^\bullet\Sigma$,

   (b) $t^\bullet \cap \Sigma^\bullet \neq \emptyset$ implies $t^\bullet \subseteq \Sigma^\bullet$, and

   (c) ${}^\bullet t \subseteq {}^\bullet\Sigma$ and $t^\bullet \subseteq \Sigma^\bullet$ together imply that ${}^\bullet t = {}^\bullet\Sigma$ if and only if $t^\bullet = \Sigma^\bullet$.

3. s has no duplicates other than itself, and if ${}^\bullet s = \emptyset$ (or $s^\bullet = \emptyset$) then $s \in {}^\bullet\Sigma$ (resp. $s \in \Sigma^\bullet$).

4. If t is not a simple transition, or if ${}^\bullet\Sigma \cup \Sigma^\bullet$ is a proper subset of ${}^\bullet t \cup t^\bullet$ then t is labelled by $\emptyset$ and there are transitions u and w in Σ with conjugate labels such that $t \bowtie_\Sigma uw$. □

5.6.1 Duplication equivalent boxes

The first goal (crucial from the point of view of dealing with axiomatisation of duplication equivalence of box expressions) is to structurally characterise the maximal synchronisation sets of boxes. To this end some auxiliary sets of transitions, called *ex*-transitions and choice context transitions are introduced.

An *ex–transition* of a box Σ is a simple transition t such that ${}^\bullet t = {}^\bullet\Sigma$ and $t^\bullet = \Sigma^\bullet$. This is denoted by $t \in \mathsf{EX}_\Sigma$ and use ex_Σ to denote the set of labels of all *ex*–transitions of Σ.[3] Choice context transitions duplicate each other except that they may have different (communication) labels. The terminology is motivated by the fact that such transitions always result from applying the choice composition operator. A set of *choice context* transitions of a box Σ is a maximal non-empty set U of transitions labelled with communication actions and all having the same connectivity. In addition, ccall_Σ is defined as

$$\mathsf{ccall}_\Sigma = \{\lambda_\Sigma(U) \mid U \text{ is a set of choice context transitions}\}$$

If the transitions in U are not *ex*-transitions then U is a set of *internal* choice context transitions and ccint_Σ is defined as

$$\mathsf{ccint}_\Sigma = \{\lambda_\Sigma(U) \mid U \text{ is a set of internal choice context transitions}\}.$$

Note that ex_Σ is a set of actions, and ccall_Σ and ccint_Σ are sets of sets of communication actions. For the boxes in Figure 5.9, $\mathsf{ex}_{\Sigma_1} = \{a\}$, $\mathsf{ccall}_{\Sigma_1} = \{\{\widehat{a},\widehat{b}\},\{\widehat{c}\},\{a\}\}$, $\mathsf{ccint}_{\Sigma_1} = \{\{\widehat{a},\widehat{b}\},\{\widehat{c}\}\}$, $\mathsf{ex}_{\Sigma_2} = \{a,b,c,\emptyset\}$, $\mathsf{ccall}_{\Sigma_2} = \{\{a,b,c\}\}$ and $\mathsf{ccint}_{\Sigma_2} = \emptyset$.

$$E_1 = ((\widehat{a} \,\Box\, \widehat{b}); (\widehat{c} \,\Box\, \emptyset)) \,\Box\, a \text{ and } E_2 = (a \,\Box\, b) \,\Box\, (c \,\Box\, \emptyset)$$

Proposition 54 Let Σ be a box and t be its transition.

1. If $t \bowtie_{\Sigma u} \delta$ for some transition u in Σ labelled by a communication action, then $\mathsf{EX}_\Sigma \neq \emptyset$.

[3]The notion of an *ex*-transition as well as other auxiliary notation introduced in this section are defined for boxes; however, they extend without any change to labelled nets.

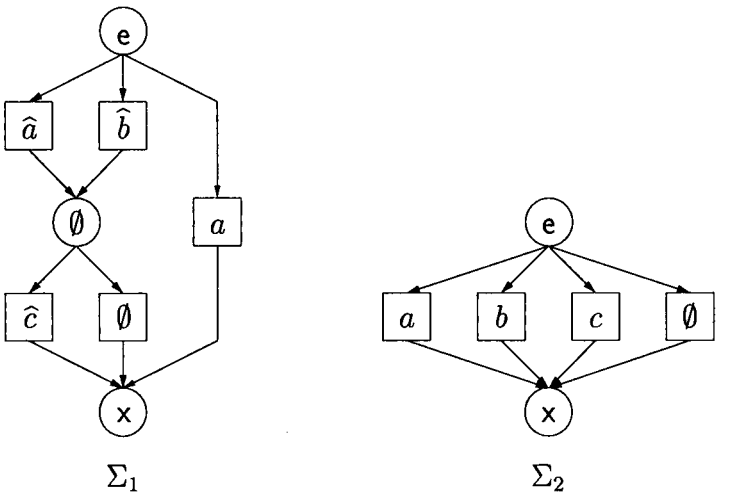


Figure 5.9: Boxes generated by E_1 and E_2

2. If ${}^\bullet t = {}^\bullet\Sigma$, $t^\bullet = \Sigma^\bullet$, $(t, {}^\bullet\Sigma) \in \mathsf{const}_\Sigma$, $(t, \Sigma^\bullet) \in \mathsf{const}_\Sigma$ and t is not an ex-transition then there are two ex-transitions u and w in Σ with conjugate labels satisfying $t \bowtie_\Sigma uw$.

Proof: (1) The result will follow if it can be shown that, for every Σ obtained from a synchronisation-free expression, the following hold: (i) it is impossible that $t \bowtie_\Sigma u\delta$; and (ii) if v, w are transitions such that $vw \bowtie_\Sigma u\delta$ then v or w or u is an ex-transition.

Firstly, it can be observed that (i) follows from the fact that all transitions in Σ derived from a synchronisation-free expression are simple. Thus $t \bowtie_\Sigma u\delta$ would imply, also due to proposition 53(2a,2b),[4] ${}^\bullet u = u^\bullet = \emptyset$, a contradiction.

Part (ii) can be proved by induction on the structure of the synchronisation-free expression E generating Σ. In the cases when $E = F;G$ or $E = F\|G$ or $E = [F * G * H]$, one can conclude that $vw \bowtie_\Sigma u\delta$ is simply impossible to satisfy. Suppose $E = F \,[]\, G$. Two cases are considered:

Case 1: $v \in \mathsf{box}(F)$ and $w \in \mathsf{box}(G)$. Also, without loss of generality, let

[4]Note that if Σ is derived from a synchronisation-free expression, then in the formulation of proposition 53(2) it does not matter whether t is labelled by a communication or internal action.

$u \in \mathsf{box}(F)$. Consider a slightly modified expression, $E' = (F \,\square\, \emptyset) \,\square\, G$. Then $vw \bowtie_{\mathsf{box}(E')} uz$ where z is the only transition of $\mathsf{box}(\emptyset)$. Hence, by proposition 41, w is an ex-transition in $\mathsf{box}(G)$ and so also in Σ.

Case 2: $v, w \in \mathsf{box}(F)$. If $u \in \mathsf{box}(F)$ then $vw \bowtie_{\mathsf{box}(F)} u\delta$ and the induction hypothesis can be used. If $u \in \mathsf{box}(G)$ then the same E' as in the first case can be used to conclude that u is an ex-transition in Σ.

(2) Since t is not a simple transition, it must have arisen as a synchronisation of two transitions u and w with conjugate labels and satisfying ${}^\bullet u \cup {}^\bullet w \subseteq {}^\bullet\Sigma$ and $u^\bullet \cup w^\bullet \subseteq \Sigma^\bullet$. Moreover, by $(t, {}^\bullet\Sigma) \in \mathsf{const}_\Sigma$ and $(t, \Sigma^\bullet) \in \mathsf{const}_\Sigma$, ${}^\bullet u = {}^\bullet w = {}^\bullet\Sigma$ and $u^\bullet = w^\bullet = \Sigma^\bullet$. Hence u and w are ex-transitions. □

The basic idea behind the structural characterisation of maximal synchronisation sets is that one can apply an a-synchronisation, without losing duplication equivalence, if for every pair of a–synchronisable transitions t and u it is possible to find a duplicate of their synchronisation in at least one of two different ways: as a syntactically generated $\emptyset$-transition, or as a synchronisation of two transitions with the same connectivity as t and u. To make the latter point explicit, suppose that $(\Sigma_1 \| \Sigma_2)$ **sy** $A \simeq (\Sigma_1 \| \Sigma_2)$ **sy** A **sy** a. Then, if t is a transition in Σ_1 and u is a transition in Σ_2 then it must be possible to find A-synchronisable transitions, t' in Σ_1 and u' in Σ_2, with the same connectivity as respectively t and u. In other words, a necessary condition for $(\Sigma_1 \| \Sigma_2)$ **sy** $A \simeq (\Sigma_1 \| \Sigma_2)$ **sy** A **sy** a to hold is that for every pair of a–synchronisable transitions t and u from respectively Σ_1 and Σ_2, there is a pair of A–synchronisable transitions, t' and u', which have the same connectivity as respectively t and u. This can be expressed rather conveniently using the sets $\mathsf{ccall}_{\Sigma_1}$ and $\mathsf{ccall}_{\Sigma_2}$ and some auxiliary notation. Let $\mathcal{Z}$ and $\mathcal{W}$ be two sets of sets of actions and A be a synchronisation set. Then $\mathsf{cov}^A(Z, W)$ is the set of all communication actions a such that if $Z \in \mathcal{Z}$ and $W \in \mathcal{W}$ satisfy $a \in Z \wedge \widehat{a} \in W$ or $\widehat{a} \in Z \wedge a \in W$ then there is $c \in A$ such that $c \in Z \wedge \widehat{c} \in W$.

Define, for a synchronisation set A and boxes Σ_1, Σ_2,

$$\mathsf{covall}^{A}_{\Sigma_1\Sigma_2} = \mathsf{cov}^{A}(\mathsf{ccall}_{\Sigma_1}, \mathsf{ccall}_{\Sigma_2}).$$

The above necessary condition for $(\Sigma_1\|\Sigma_2)$ **sy** $A \simeq (\Sigma_1\|\Sigma_2)$ **sy** A **sy** a simply amounts to saying that a and $\hat{a}$ belong to $\mathsf{covall}^{A}_{\Sigma_1\Sigma_2}$. Note that for the nets in Figure 5.9, $\mathsf{covall}^{\mathsf{a}}_{\Sigma_1\Sigma_2} = \mathcal{A} - \mathsf{c}$. Characterising maximal synchronisation sets is rather straightforward in the case of sequential, parallel and iteration composition.

Proposition 55 Let Σ_i $(i = 1, 2, 3)$ be boxes and A be a synchronisation set. Then

$$\begin{aligned} \mathsf{max}_{(\Sigma_1;\Sigma_2)\ \mathbf{sy}\ A} &= \mathsf{covall}^{A}_{\Sigma_1\Sigma_2} \cap \bigcap_{i=1}^{2} \mathsf{max}_{\Sigma_i\ \mathbf{sy}\ A} \\ \mathsf{max}_{(\Sigma_1\|\Sigma_2)\ \mathbf{sy}\ A} &= \mathsf{covall}^{A}_{\Sigma_1\Sigma_2} \cap \bigcap_{i=1}^{2} \mathsf{max}_{\Sigma_i\ \mathbf{sy}\ A} \\ \text{and}\quad \mathsf{max}_{[\Sigma_1*\Sigma_2*\Sigma_3]\ \mathbf{sy}\ A} &= \mathsf{covall}^{A}_{\Sigma_1\Sigma_2\Sigma_3} \cap \bigcap_{i=1}^{3} \mathsf{max}_{\Sigma_i\ \mathbf{sy}\ A}. \end{aligned}$$

Proof: Only the most complicated case is considered, viz. iteration. To show the $(\subseteq)$ inclusion, suppose that $a \notin A$ (clearly, if $a \in A$ then $a \in \mathsf{covall}^{A}_{\Sigma_1\Sigma_2\Sigma_3}$ and $a \in \mathsf{max}_{\Sigma_i\ \mathbf{sy}\ A}$, for $i = 1, 2, 3$) and

$$[\Sigma_1 * \Sigma_2 * \Sigma_3]\ \mathbf{sy}\ (A \cup \mathsf{a}) \simeq [\Sigma_1 * \Sigma_2 * \Sigma_3]\ \mathbf{sy}\ A.$$

Let $\Sigma = \Sigma_1 : \{({}^\bullet\Sigma_1, {}^\bullet\Sigma_1', \mathsf{e}), (\Sigma_1{}^\bullet, {}^\bullet\Sigma_2, \Sigma_2'{}^\bullet, {}^\bullet\Sigma_3, \emptyset)\}$ where each Σ_i' is a copy of Σ_i (in the same way as in the definition of the iteration operator). Moreover, let $\Sigma' = (\Sigma_2, \Sigma_3, \Sigma_1', \Sigma_2', \Sigma_3') : \Delta$ be the union of all the remaining nets in the definition of $[\Sigma_1 * \Sigma_2 * \Sigma_3]$. It will be shown that $a \in \mathsf{max}_{\Sigma_i\ \mathbf{sy}\ A}$, for $i = 1, 2, 3$, and $a \in \mathsf{covall}^{A}_{\Sigma_1\Sigma_2\Sigma_3}$ using the results and notation from section 5.4.2.

Let $E = {}^\bullet\Sigma_1 \otimes {}^\bullet\Sigma_1'$ and $X = \Sigma_1{}^\bullet \otimes {}^\bullet\Sigma_2 \otimes \Sigma_2'{}^\bullet \otimes {}^\bullet\Sigma_3$. It can be observed that A1-A5 hold by the definition of the iteration operator and proposition 53(1,2). To show that A6 also holds, suppose that w is an EX-transition in $(\Sigma \cup \Sigma')$ **sy** A. It is easy to see that then w belongs to Σ **sy** A because every transition in Σ' is connected to at least one place not in $E \cup X$.

Since A1-A6 hold, by proposition 51, $a \in \mathsf{max}_{\Sigma}\ \mathbf{sy}_{\ A}$. Moreover, $\Sigma \simeq \Sigma_a$, where Σ_a is Σ_1 with the label of each exit place changed to $\emptyset$. Hence $a \in \mathsf{max}_{\Sigma_a}\ \mathbf{sy}_{\ A}$. Therefore, by proposition 53(3), $[\Sigma_a]_{\simeq}$ is $[\Sigma_1]_{\simeq}$ with the label of every exit place changed to $\emptyset$. Hence $a \in \mathsf{max}_{\Sigma_1}\ \mathbf{sy}_{\ A}$. In a similar way, one may show that $a \in \mathsf{max}_{\Sigma_2}\ \mathbf{sy}_{\ A}$ and $a \in \mathsf{max}_{\Sigma_3}\ \mathbf{sy}_{\ A}$.

To show that $a \in \mathsf{covall}^{A}_{\Sigma_1\Sigma_2\Sigma_3}$ suppose that $(t, v) \in (T_{\Sigma} \times T_{\Sigma'}) \cap \mathsf{syn}_{\mathsf{a}}$. It suffices to show that there are $(u, w) \in (T_{\Sigma} \times T_{\Sigma'}) \cap \mathsf{syn}_{A}$ such that $t \bowtie_{\Sigma} u$ and $v \bowtie_{\Sigma'} w$ (note that the operations of place multiplication and addition do not affect the relationship of having the same connectivity, and that by construction of the iteration operator it is not possible for two transitions coming from different nets forming Σ' to have the same connectivity in Σ'). To be able to apply proposition 52, it is first observed that $v \notin T_{EX}$ since no transition in Σ' is an EX-transition. Suppose then that $t \in T_{EX}$ and $y, z \in T_{\Sigma'}$ are such that $tv \bowtie_{\Sigma\cup\Sigma'} yz$. Then, without loss of generality, $y \in T_{\Sigma'_1}$ and $z \in T_{\Sigma'_2}$. Consequently, there are $s \in S_{\Sigma'_1} - {}^{\bullet}\Sigma'_1$ and $r \in S_{\Sigma'_2} - \Sigma'^{\bullet}_2$ such that $s \in {}^{\bullet}v$ and $p \in v^{\bullet}$, a contradiction. If it is assumed that $tv \bowtie_{\Sigma\cup\Sigma'} y$ then a contradiction follows in a similar way. Hence there is no $x \in T_{\Sigma'}\ \mathbf{sy}_{\ A}$ such that $t \diamond v \simeq_{(\Sigma\cup\Sigma')}\ \mathbf{sy}_{\ (A\cup\mathsf{a})}\ x$. Hence proposition 52 can be applied.

Thus the $(\subseteq)$ inclusion holds. The reverse one follows from proposition 50. □

Note that by setting $A = \emptyset$ it is immediately shown that, e.g., $\mathsf{max}_{\Sigma_1;\Sigma_2}$ is the set of all $a \in \mathsf{max}_{\Sigma_1}\ \mathbf{sy}_{\ A} \cap \mathsf{max}_{\Sigma_2}\ \mathbf{sy}_{\ A}$ such that if a transition labelled a or $\hat{a}$ appears in Σ_1 then there is no transition with the conjugate label in Σ_2.

A similarly pleasant characterisation does not hold for the choice composition. One of the reasons is that a synchronisation of ex–transitions can sometimes be obtained by a syntactically introduced $\emptyset$-transition. For example, if $\Sigma_1 = \mathsf{box}(a\|\hat{a})$ and $\Sigma_2 = \mathsf{box}(\emptyset)$ then $a \in \mathsf{max}_{(\Sigma_1 \square \Sigma_2)}\ \mathbf{sy}_{\ \emptyset}$ but $a \notin \mathsf{max}_{\Sigma_1}\ \mathbf{sy}_{\ \emptyset}$. Another example is provided by the boxes Σ_1 and Σ_2 in Figure 5.9 for which $a \in \mathsf{max}_{(\Sigma_1 \square \Sigma_2)}\ \mathbf{sy}\ \mathsf{b}$ but $a \notin \mathsf{max}_{\Sigma_1}\ \mathbf{sy}\ \mathsf{b} = \mathsf{max}_{\Sigma_1}$. Note that if the previous discussion were repeated for $(\Sigma_1 \square \Sigma_2)\ \mathbf{sy}\ A \simeq (\Sigma_1 \square \Sigma_2)\ \mathbf{sy}\ A\ \mathbf{sy}\ \mathsf{a}$ then it

would no longer be true that t' and u' had to have the same connectivity as t and u if, e.g., u is an ex–transition in Σ_2 since in such a case an ex–transition u' in Σ_1 could provide a suitable 'match' for t'.

The characterisation of the maximal synchronisation sets for the choice composition is more complicated. For a box Σ_1, let $\mathcal{U}$ be the set of all sets of internal choice context transitions U such that if $t \in U$ then there is no transition u in Σ_1 satisfying $u \bowtie_{\Sigma_1} \delta t$. Intuitively, this means that if t were to be synchronised with a conjugate ex-transition coming from the box Σ_2 in the context $\Sigma_1 \,\square\, \Sigma_2$ then the resulting transition would not have the same connectivity as any of the transitions present in Σ_1. $\mathsf{ccnoex}_{\Sigma_1}$ is defined as follows $\mathsf{ccnoex}_{\Sigma_1} = \{\lambda_{\Sigma_1}(U) \mid U \in \mathcal{U}\}$ and, for all boxes Σ_1 and Σ_2 and a synchronisation set A,

$$\begin{array}{lcl}
\mathsf{covnoex}^A_{\Sigma_1\Sigma_2} & = & \mathsf{cov}^A(\mathsf{ccnoex}_{\Sigma_1} \textbf{ sy } A, \{\mathsf{ex}_{\Sigma_2}\}) \;\cap \\
 & & \mathsf{cov}^A(\mathsf{ccnoex}_{\Sigma_2} \textbf{ sy } A, \{\mathsf{ex}_{\Sigma_1}\}) \\
\mathsf{covmix}^A_{\Sigma_1\Sigma_2} & = & \mathsf{cov}^A(\mathsf{ccint}_{\Sigma_1}, \mathsf{ccall}_{\Sigma_2}) \;\cap\; \mathsf{cov}^A(\mathsf{ccint}_{\Sigma_2}, \mathsf{ccall}_{\Sigma_1}) \\
\mathsf{covint}^A_{\Sigma_1\Sigma_2} & = & \mathsf{cov}^A(\mathsf{ccint}_{\Sigma_1}, \mathsf{ccint}_{\Sigma_2}).
\end{array}$$

The above definitions closely follow that of covall. For the nets in Figure 5.9, $\mathsf{ccnoex}_{\Sigma_1} = \{\{\widehat{a}, \widehat{b}\}, \{\widehat{c}\}\}$, $\mathsf{covnoex}^{\mathsf{a}}_{\Sigma_1\Sigma_2} = \mathcal{A} - \mathsf{c}$, $\mathsf{covmix}^{\mathsf{b}}_{\Sigma_1\Sigma_2} = \mathcal{A} - \mathsf{c}$ and $\mathsf{covint}^{\emptyset}_{\Sigma_1\Sigma_2} = \mathcal{A}$.

A syntactic restriction on the type of expressions used to derive boxes is introduced. Let Exp_0 denote those box expressions E for which there is no subexpression $F \,\square\, G$ and a communication action a such that $a \in \mathsf{ex}_{\mathsf{box}(F)}$ and $\widehat{a} \in \mathsf{ex}_{\mathsf{box}(G)}$. Let Box_0 denote boxes which can be derived from the box expressions in Exp_0.

Proposition 56 Let Σ be a box in Box_0.

1. There are no transitions t and u in Σ with conjugate labels and satisfying ${}^\bullet t = {}^\bullet u$ and $t^\bullet = u^\bullet$.

2. There is no $\emptyset$-labelled transition in Σ such that all the arcs adjacent to it have weight 2.

Proof: (1) By straightforward induction on the structure of the expression generating boxes in Box_0.

(2) Follows immediately from the first part. □

A partial characterisation of the maximal synchronisation sets of boxes involving choice composition is then obtained.

Proposition 57 Let $\Sigma = (\Sigma_1 \,\square\, \Sigma_2)$ **sy** A be a box in Box_0 and A be a synchronisation set such that

$$\mathsf{ex}_\Sigma \cap (A \cup \{\emptyset\}) \;\subseteq\; \mathsf{ex}_{\Sigma_1} \textbf{ sy } _A \cap \mathsf{ex}_{\Sigma_2} \textbf{ sy } _A. \tag{5.3}$$

Then

$$\mathsf{max}_\Sigma \;=\; \mathsf{covnoex}^A_{\Sigma_1\Sigma_2} \cap \mathsf{covint}^A_{\Sigma_1\Sigma_2} \cap \bigcap_{i=1}^{2} \mathsf{max}_{\Sigma_i} \textbf{ sy } _A.$$

Proof: Using notation from section 5.4.2, let $E = {}^\bullet\Sigma_a = {}^\bullet\Sigma_b$ and $X = \Sigma_a{}^\bullet = \Sigma_b{}^\bullet$, where

$$\begin{aligned} \Sigma_a &= \Sigma_1 : \{({}^\bullet\Sigma_1, {}^\bullet\Sigma_2, \mathsf{e}), (\Sigma_1{}^\bullet, \Sigma_2{}^\bullet, \mathsf{x})\} \\ \Sigma_b &= \Sigma_2 : \{({}^\bullet\Sigma_1, {}^\bullet\Sigma_2, \mathsf{e}), (\Sigma_1{}^\bullet, \Sigma_2{}^\bullet, \mathsf{x})\}. \end{aligned}$$

Note that $\Sigma_1 \,\square\, \Sigma_2 = \Sigma_a \cup \Sigma_b$.

Suppose that $a \in \mathsf{max}_\Sigma$ and $a \notin A$ (if $a \in A$ then a clearly belongs to the rhs of the equality from the formulation of this proposition). Using proposition 53(1,2) and the definition of the choice operator, one can see that A1–A5 are satisfied. Moreover, A6 holds which follows from (5.3) and $T_{EX} \cap T_{\Sigma'} = \mathsf{EX}_{\Sigma'}$, for $\Sigma' \in \{\Sigma, \Sigma_1 \textbf{ sy } A, \Sigma_2 \textbf{ sy } A\}$ (note that the latter follows from propositions 53(1) and 56(2)). Hence, by proposition 51, $a \in \mathsf{max}_{\Sigma_a} \textbf{ sy } _A$. Since $\Sigma_a \simeq \Sigma_1$, $a \in \mathsf{max}_{\Sigma_1} \textbf{ sy } _A$ and, by symmetry, $a \in \mathsf{max}_{\Sigma_2} \textbf{ sy } _A$.

To show $a \in \mathsf{covint}^A_{\Sigma_a\Sigma_b} = \mathsf{covint}^A_{\Sigma_1\Sigma_2}$ it needs to be shown that if $(t, v) \in ((T_{\Sigma_a} - \mathsf{EX}_{\Sigma_a}) \times (T_{\Sigma_b} - \mathsf{EX}_{\Sigma_b})) \cap \mathsf{syn_a}$ then there are $(u, w) \in (T_{\Sigma_a} \times T_{\Sigma_b}) \cap \mathsf{syn}_A$ such that $t \bowtie_{\Sigma_a} u$ and $v \bowtie_{\Sigma_b} w$. This, however, follows directly from proposition 52 (note that $t, v \notin T_{EX}$).

To show $a \in \mathsf{covnoex}^A_{\Sigma_a\Sigma_b} = \mathsf{covnoex}^A_{\Sigma_1\Sigma_2}$, by symmetry, it is sufficient to prove that if $t \in \mathsf{EX}_{\Sigma_a} = \mathsf{EX}_{\Sigma_1}$ is an a-labelled transition, and $v \in U \in \mathsf{ccnoex}_{\Sigma_b}\ \mathbf{sy}_A = \mathsf{ccnoex}_{\Sigma_2}\ \mathbf{sy}_A$ is an $\widehat{a}$-labelled transition then there are $(u, w) \in (T_{\Sigma_a} \times T_{\Sigma_b}) \cap \mathsf{syn}_A$ such that $t \bowtie_{\Sigma_a} u$ and $v \bowtie_{\Sigma_b} w$. This, however, follows directly from proposition 52 (the proposition can be applied since $v \notin T_{EX}$, and if $t \in T_{EX}$ and $t \diamond v \simeq_\Sigma \mathbf{sy}_{(A \cup \mathsf{a})}\ x$ where $x \in T_{\Sigma_b}\ \mathbf{sy}_A$ then $v\delta \bowtie_{\Sigma_b} \mathbf{sy}_A\ x$, contradicting the choice of v).

The $(\subseteq)$ inclusion has been shown. To show the reverse one, corollary 8 is used. That it can be applied is shown by the following argument. Suppose that $(t, v) \in (T_{\Sigma_a} \times T_{\Sigma_b}) \cap \mathsf{syn}_{\mathsf{a}}$, where a belongs to the rhs of the equality from the formulation of this proposition. Three cases are considered.

Case 1: $t \notin \mathsf{EX}_{\Sigma_a}$ and $v \notin \mathsf{EX}_{\Sigma_b}$. By $a \in \mathsf{covint}^A_{\Sigma_1\Sigma_2}$, there are $(u, w) \in (T_{\Sigma_a} \times T_{\Sigma_b}) \cap \mathsf{syn}_A$ such that $t \bowtie_{\Sigma_a} u$ and $v \bowtie_{\Sigma_b} w$. Hence $tv \bowtie_{\Sigma_a \cup \Sigma_b} uw$.

Case 2: $t \in \mathsf{EX}_{\Sigma_a}$ and $v \notin \mathsf{EX}_{\Sigma_b}$ (note the argument is symmetric for $t \notin \mathsf{EX}_{\Sigma_a}$ and $v \in \mathsf{EX}_{\Sigma_b}$). If it is not the case that there is $z \in T_{\Sigma_b}\ \mathbf{sy}_A$ such that $tv \simeq_\Sigma z$ then there is $U \in \mathsf{ccnoex}_{\Sigma_b}\ \mathbf{sy}_A$ such that $t \in U$ and the fact that $a \in \mathsf{covnoex}^A_{\Sigma_1\Sigma_2}$ can be used to reach the desired conclusion, i.e. to find u and w as in Case 1.

Case 3: $t \in \mathsf{EX}_{\Sigma_a}$ and $v \in \mathsf{EX}_{\Sigma_b}$. This produces a contradiction with proposition 56. □

Finally, it is observed that the various sets of communication actions introduced and used in this section are preserved through duplication equivalence and synchronisation.

Proposition 58 Let A, B, C be synchronisation sets and Σ_a, Σ_b, Σ_c, Σ_d be boxes such that $\Sigma_a \simeq \Sigma_c$ and $\Sigma_b \simeq \Sigma_d$.

1. $\mathsf{set}_{\Sigma_a} = \mathsf{set}_{\Sigma_c}$ for $\mathsf{set} \in \{\mathsf{ex}, \mathsf{ccall}, \mathsf{ccint}, \mathsf{ccnoex}\}$.
2. $\mathsf{set}^A_{\Sigma_a\Sigma_b} = \mathsf{set}^A_{\Sigma_c\Sigma_d}$ for $\mathsf{set} \in \{\mathsf{covall}, \mathsf{covmix}, \mathsf{covint}, \mathsf{covnoex}\}$.
3. $\mathsf{set}^A_{\Sigma_a\Sigma_b} = \mathsf{set}^A_{\Sigma_a}\ \mathbf{sy}_{B,\Sigma_b}\ \mathbf{sy}_C$ for $\mathsf{set} \in \{\mathsf{covall}, \mathsf{covmix}, \mathsf{covint}\}$. □

Note that $\mathsf{covnoex}^{A}_{\Sigma_a\Sigma_b} = \mathsf{covnoex}^{A}_{\Sigma_a}$ **sy** $_{B,\Sigma_b}$ **sy** $_{C}$ does not, in general, hold. For example, if $\Sigma_a = \mathsf{box}(\widehat{a})$ and $\Sigma_b = \mathsf{box}(\widehat{a} \,\square\, (a;\emptyset))$ then $\mathsf{covnoex}^{\emptyset}_{\Sigma_a,\Sigma_b}$ **sy** $\mathsf{a} = \mathcal{A}$ but $a \notin \mathsf{covnoex}^{\emptyset}_{\Sigma_a\Sigma_b}$.

5.7 Box expressions

The notion of duplication equivalence formulated for boxes is now transferred to the domain of box expressions. Two box expressions, E and F, are *duplication equivalent* if $\mathsf{box}(E) \simeq \mathsf{box}(F)$. This is denoted by $E \simeq F$. The maximal synchronisation set of a box expression E is defined as $\mathsf{max}_E = \mathsf{max}_{\mathsf{box}(E)}$. Clearly, many properties of duplication equivalence that hold for boxes can be transferred to box expressions. In particular, it is immediately obtained that $\simeq$ is a congruence in the domain of box expressions.

Moreover, directly from propositions 46 and 47:

Proposition 59 Let E and F be box expressions, and A and B be synchronisation sets.

1. E **sy** A **sy** $B \simeq E$ **sy** $(A \cup B)$.
2. If $E \simeq E$ **sy** B and $A \subseteq B$ then $E \simeq E$ **sy** A.
3. If $E \simeq E$ **sy** A and $E \simeq E$ **sy** B then $E \simeq E$ **sy** $(A \cup B)$.
4. If $E \simeq F$ **sy** A then $E \simeq E$ **sy** A. □

The main aim of this section is to axiomatise duplication equivalence of box expressions. A crucial difficulty to be solved is the development of a structural characterisation of maximal synchronisation sets, both in order to obtain a set of sound axioms and to define normal form box expressions needed for a completeness proof. Such a characterisation is based on that obtained for boxes; Therefore, the expression counterparts of ex–transitions and choice context transitions as well as other notations introduced in the previous section

are defined. For a box expression E, let ex_E and potex_E be sets of actions defined by induction on the structure of E, as follows:

$$\begin{array}{lcllcl} \mathsf{ex}_\alpha & = & \{\alpha\} & \mathsf{potex}_{E\|F} & = & (\mathsf{ex}_E \cap \widehat{\mathsf{ex}_F}) \cup (\mathsf{ex}_F \cap \widehat{\mathsf{ex}_E}) \\ \mathsf{ex}_{E\,\square\,F} & = & \mathsf{ex}_E \cup \mathsf{ex}_F & \mathsf{potex}_{E\,\square\,F} & = & \mathsf{potex}_E \cup \mathsf{potex}_F \\ & & & \mathsf{potex}_{E\ \mathbf{sy}\ A} & = & \mathsf{potex}_E \end{array}$$

$$\mathsf{ex}_{E\ \mathbf{sy}\ A} = \begin{cases} \mathsf{ex}_E \cup \{\emptyset\} & \text{if } \mathsf{potex}_E \cap A \neq \emptyset \\ \mathsf{ex}_E & \text{otherwise} \end{cases}$$

In all the remaining cases, ex_E and potex_E are defined as empty. The meaning of ex_E is that of $\mathsf{ex}_{\mathsf{box}(E)}$. The auxiliary set potex_E represents *potential* $\emptyset$-labelled ex-transitions which can be generated by applying synchronisation using the actions in potex_E. For example, $\mathsf{potex}_{(a\,\square\,b)\|(\widehat{a}\,\square\,\widehat{b})} = \mathsf{a} \cup \mathsf{b}$.

Proposition 60 For every box expression E, $\mathsf{ex}_E = \mathsf{ex}_{\mathsf{box}(E)}$ and $\mathsf{potex}_E = \lambda_{\mathsf{box}(E)}(T_0)$, where T_0 is the set of all transitions t, u in $\mathsf{box}(E)$ labelled with conjugate communication actions such that $tu \bowtie_{\mathsf{box}(E)} \delta$.

Proof: By induction on the structure of E. □

Choice context transitions are now considered. For a box expression E, let ccint_E and ccall_E be two sets of sets of communication actions defined by induction on the structure of E, as follows:

$$\begin{array}{lcllcl} \mathsf{ccint}_\alpha & = & \emptyset & \mathsf{ccint}_{[E*F*G]} & = & \mathsf{ccall}_E \cup \mathsf{ccall}_F \cup \mathsf{ccall}_G \\ \mathsf{ccint}_{E\,\square\,F} & = & \mathsf{ccint}_E \cup \mathsf{ccint}_F & \mathsf{ccint}_{E\|F} & = & \mathsf{ccall}_E \cup \mathsf{ccall}_F \\ \mathsf{ccint}_{E;F} & = & \mathsf{ccall}_E \cup \mathsf{ccall}_F & \mathsf{ccint}_{E\ \mathbf{sy}\ A} & = & \mathsf{ccint}_E \end{array}$$

$$\mathsf{ccall}_E = \begin{cases} \mathsf{ccint}_E \cup \{\mathsf{ex}_E \cap \mathcal{A}\} & \text{if } \mathsf{ex}_E \cap \mathcal{A} \neq \emptyset \\ \mathsf{ccint}_E & \text{otherwise.} \end{cases}$$

Moreover, a set of sets of communication actions ccnoex_E, is defined as follows:

$$\begin{array}{rcl} \mathsf{ccnoex}_{F\,\square\,G} & = & \mathsf{ccnoex}_F \cup \mathsf{ccnoex}_G \\ \text{and } \mathsf{ccnoex}_{F\ \mathbf{sy}\ A} & = & \{C \in \mathsf{ccnoex}_F \mid \widehat{C} \cap \mathsf{ex}_F \cap A = \emptyset\} \end{array}$$

and by setting $\mathsf{ccnoex}_E = \mathsf{ccint}_E$ in all the remaining cases.

Proposition 61 For every box expression E, $\mathsf{ccall}_E = \mathsf{ccall}_{\mathsf{box}(E)}$, $\mathsf{ccint}_E = \mathsf{ccint}_{\mathsf{box}(E)}$ and $\mathsf{ccnoex}_E = \mathsf{ccnoex}_{\mathsf{box}(E)}$.

Proof: By induction on the structure of E and using proposition 60. □

Let E, F and G be box expressions and A be a synchronisation set. Then:

$$\begin{array}{lcl}
\mathsf{covall}^A_{EF} & = & \mathsf{cov}^A(\mathsf{ccall}_E, \mathsf{ccall}_F) \\
\mathsf{covnoex}^A_{EF} & = & \mathsf{cov}^A(\mathsf{ccnoex}_E \ \mathbf{sy}_{\ A}, \{\mathsf{ex}_F\}) \ \cap \ \mathsf{cov}^A(\mathsf{ccnoex}_F \ \mathbf{sy}_{\ A}, \{\mathsf{ex}_E\}) \\
\mathsf{covmix}^A_{EF} & = & \mathsf{cov}^A(\mathsf{ccint}_E, \mathsf{ccall}_F) \ \cap \ \mathsf{cov}^A(\mathsf{ccint}_F, \mathsf{ccall}_E) \\
\mathsf{covint}^A_{EF} & = & \mathsf{cov}^A(\mathsf{ccint}_E, \mathsf{ccint}_F)
\end{array}$$

and $\mathsf{covall}^A_{EFG} = \bigcap\{\mathsf{covall}^A_{\mathcal{XY}} \mid \mathcal{X}, \mathcal{Y} \in \{E, F, G\}\}$. The sets of communication actions that have just been defined are direct counterparts of similar notions introduced for boxes.

Proposition 62 Let E, F and G be expressions and A be a synchronisation set.

1. $\mathsf{set}^A_{EF} = \mathsf{set}^A_{\mathsf{box}(E)\mathsf{box}(F)}$, for $\mathsf{set} \in \{\mathsf{covall}, \mathsf{covint}, \mathsf{covmix}, \mathsf{covnoex}\}$.

2. $\mathsf{covall}^A_{EFG} = \mathsf{covall}^A_{\mathsf{box}(E)\mathsf{box}(F)\mathsf{box}(G)}$.

Proof: Follows from propositions 60 and 61. □

As a result, the relationship between max_E and the structure of E can be captured by adapting the results obtained for boxes. Below, for a pair of expressions E and F, simex_{EF} denotes the set of all synchronisation sets A such that

$$\mathsf{ex}_{(E \,\square\, F)} \ \mathbf{sy}_{\ A} \cap (A \cup \{\emptyset\}) \ \subseteq \ \mathsf{ex}_E \ \mathbf{sy}_{\ A} \cap \mathsf{ex}_F \ \mathbf{sy}_{\ A}.$$

Proposition 63 Let E_i $(i = 1, 2, 3)$ be box expressions and A be a synchronisation set. Then

$$\begin{aligned} \mathsf{max}_{(E_1;E_2)}\ \mathbf{sy}\ A &= \mathsf{covall}^A_{E_1E_2} \cap \bigcap_{i=1}^{2} \mathsf{max}_{E_i}\ \mathbf{sy}\ A \\ \mathsf{max}_{(E_1\|E_2)}\ \mathbf{sy}\ A &= \mathsf{covall}^A_{E_1E_2} \cap \bigcap_{i=1}^{2} \mathsf{max}_{E_i}\ \mathbf{sy}\ A \\ \text{and}\quad \mathsf{max}_{[E_1*E_2*E_3]}\ \mathbf{sy}\ A &= \mathsf{covall}^A_{E_1E_2E_3} \cap \bigcap_{i=1}^{3} \mathsf{max}_{E_i}\ \mathbf{sy}\ A. \end{aligned}$$

Moreover, if $(E_1 \,[]\, E_2)$ **sy** $A \in \mathsf{Exp}_0$ and $A \in \mathsf{simex}_{E_1E_2}$ then

$$\mathsf{max}_{(E_1 \,[]\, E_2)}\ \mathbf{sy}\ A = \mathsf{covnoex}^A_{E_1E_2} \cap \mathsf{covint}^A_{E_1E_2} \cap \bigcap_{i=1}^{2} \mathsf{max}_{E_i}\ \mathbf{sy}\ A.$$

Proof: Follows from propositions 55, 57 and 62. □

The above characterisation of maximal synchronisation sets of expressions involving the choice operator is only partial. For that reason, two results on specific usages of choice composition are now provided.

Proposition 64 Let $E = (E_1\|E_2)$ **sy** A be a box expression in Exp_0 and a be a communication action such that $a \in \mathsf{max}_{E\,[]\,\emptyset} - \mathsf{max}_E$. Then $\mathsf{ex}_{E_1} \cap \widehat{\mathsf{ex}_{E_2}} \cap \mathsf{a} \neq \emptyset$ and

$$a \in \mathsf{covmix}^A_{E_1E_2} \cap \bigcap_{i=1}^{2} \mathsf{max}_{E_i}\ \mathbf{sy}\ A.$$

Proof: Let $\Sigma = \mathsf{box}(\emptyset)$ and $\Sigma_i = \mathsf{box}(E_i)$, for $i = 1, 2$. Since $|{}^\bullet\Sigma| = |\Sigma^\bullet| = 1$, it may be assumed that $(\Sigma_1\|\Sigma_2) \,[]\, \Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$ where

$$\Sigma_3 = \Sigma \oplus \{(s, {}^\bullet\Sigma_1 \cup {}^\bullet\Sigma_2, \mathsf{e}), (r, \Sigma_1{}^\bullet \cup \Sigma_2{}^\bullet, \mathsf{x})\}$$

and s and r are places satisfying $\{s\} = {}^\bullet\Sigma$ and $\{r\} = \Sigma^\bullet$. Therefore

$$\begin{aligned} (\Sigma_1 \cup \Sigma_2)\ \mathbf{sy}\ A &\not\simeq (\Sigma_1 \cup \Sigma_2)\ \mathbf{sy}\ (A \cup \mathsf{a}) \\ \text{and}\quad (\Sigma_1 \cup \Sigma_2 \cup \Sigma_3)\ \mathbf{sy}\ A &\simeq (\Sigma_1 \cup \Sigma_2 \cup \Sigma_3)\ \mathbf{sy}\ (A \cup \mathsf{a}) \end{aligned}$$

Since $(\Sigma_1 \cup \Sigma_2)$ **sy** A and $(\Sigma_1 \cup \Sigma_2 \cup \Sigma_3)$ **sy** A differ only by a single $\emptyset$–labelled transition $t \in \mathsf{EX}_{\Sigma_1\cup\Sigma_2\cup\Sigma_3}$, from proposition 43(2) it follows that there must be a–synchronisable transitions u, w in $\Sigma_1 \cup \Sigma_2$ such that $\delta \bowtie_{\Sigma_1\cup\Sigma_2} uw$. Hence $\mathsf{ex}_{E_1} \cap \widehat{\mathsf{ex}_{E_2}} \cap \mathsf{a} \neq \emptyset$.

To prove the second part, the results and notation from section 5.4.2, are used with boxes Σ_1 and $\Sigma_2 \cup \Sigma_3$, and sets of places $E = {}^\bullet\Sigma_1$ and $X = \Sigma_1{}^\bullet$. It is easy to see, using proposition 53, that A1-A5 hold. Moreover, the only EX-transitions (if any) in $(\Sigma_1 \cup (\Sigma_2 \cup \Sigma_3))$ **sy** A are those in Σ_1 **sy** A. Hence A6 also holds and, by proposition 51, $a \in \mathsf{max}_{E_1}$ **sy** $_A$ is obtained and, by symmetry, $a \in \mathsf{max}_{E_2}$ **sy** $_A$.

It can also be observed that proposition 52 can be applied for $(t, v) \in ((T_{\Sigma_1} - \mathsf{EX}_{\Sigma_1}) \times T_{\Sigma_2 \cup \Sigma_3}) \cap \mathsf{syn_a}$ since then $t, v \notin T_{EX}$. From this, and by symmetry, $a \in \mathsf{covmix}^A_{E_1 E_2}$. □

Proposition 65 Let $a \in \mathcal{A}$ be such that $a \in \mathsf{max}_{(E \,\square\, \alpha_1 \,\square \ldots \square\, \alpha_k)}$ **sy** $_A$ where $\alpha_1, \ldots \alpha_k$ are actions and $E \,\square\, \alpha_1 \,\square \ldots \,\square\, \alpha_k$ is an expression in Exp_0.

1. If the topmost operator of E is sequence or iteration then $a \in \mathsf{max}_E$.
2. If the topmost operator of E is parallel composition and $a \notin \mathsf{max}_E$ then $a \in \mathsf{max}_{(E \text{ sy } A \,\square\, \emptyset)}$ and $\alpha_i = \emptyset$, for some $i \leq k$.

Proof: Let $\Sigma = \mathsf{box}(E)$ and $\Sigma_i = \mathsf{box}(\alpha_i)$ for $i = 1, \ldots, k$. Since $|{}^\bullet\Sigma_i| = |\Sigma_i{}^\bullet| = 1$ for every $i \leq k$, it may be assumed that

$$\Sigma_a = \mathsf{box}(E \,\square\, \alpha_1 \,\square \ldots \,\square\, \alpha_k) = \Sigma \cup \Sigma'_1 \cup \ldots \cup \Sigma'_k$$

where in the above, for $i = 1, \ldots, k$,

$$\Sigma'_i = \Sigma_i \oplus \{(s_i, {}^\bullet\Sigma, \mathsf{e}), (r_i, \Sigma^\bullet, \mathsf{x})\}$$

and s_i and r_i are places satisfying $\{s_i\} = {}^\bullet\Sigma_i$ and $\{r_i\} = \Sigma_i{}^\bullet$.

By proposition 47(3), Σ_a **sy** $A \cong \Sigma_a$ **sy** $(A \cup \mathsf{a})$. If $a \notin \mathsf{max}_E$ then, by proposition 43(1,2) and $a \in \mathsf{max}_{(E \,\square\, \alpha_1 \,\square \ldots \square\, \alpha_k)}$ **sy** $_A$, there are $t, u \in T_\Sigma$ such that $(t, u) \in \mathsf{syn_a}$ and one of the following holds:

(a) There is $w \in T_{\Sigma'_1 \cup \ldots \cup \Sigma'_k}$ such that $t \diamond u \simeq_{\Sigma_a}$ **sy** $_{(A \cup \mathsf{a})}$ w.

(b) There are $v \in T_\Sigma$ and $w \in T_{\Sigma'_1 \cup \ldots \cup \Sigma'_k}$ such that $t \diamond u \simeq_{\Sigma_a}$ **sy** $_{(A \cup \mathsf{a})}$ $v \diamond w$.

(c) There are $v, w \in T_{\Sigma'_1 \cup \ldots \cup \Sigma'_k}$ such that $t \diamond u \simeq_{\Sigma_a} \mathbf{sy}_{(A \cup \mathsf{a})}\, v \diamond w$.

Clearly, (c) is impossible because the weights of all the arcs adjacent to $v \diamond w$ would be equal to 2, contradicting proposition 56(2). Three cases are considered.

Case 1: $E = F; G$. Then (a) leads to a contradiction since $t \diamond u$ is connected to at least one internal place in Σ_a **sy** $(A \cup \mathsf{a})$. That (b) also leads to a contradiction can be shown in the following way. Since ${}^\bullet w = {}^\bullet\Sigma$ and $w^\bullet = \Sigma^\bullet$ it must be the case that, without loss of generality, t is a transition from $\mathsf{box}(F)$ and u is a transition from $\mathsf{box}(G)$. Then, again without loss of generality, v is a transition from $\mathsf{box}(F)$. Note now that $\mathsf{box}(E)$ is the union of $\Sigma_f = \mathsf{box}(F) : \{(\mathsf{box}(F)^\bullet, {}^\bullet\mathsf{box}(G), \emptyset)\}$ and $\Sigma_g = \mathsf{box}(G) : \{(\mathsf{box}(F)^\bullet, {}^\bullet\mathsf{box}(G), \emptyset)\}$ and observe that there is a non-entry place s in $\mathsf{box}(E)$ which is also in ${}^\bullet u$. Hence, s also has to be in ${}^\bullet v$. But the only places shared by Σ_f and Σ_g are those in $\mathsf{box}(F)^\bullet \otimes {}^\bullet\mathsf{box}(G)$. This means that $\mathsf{box}(F)^\bullet \cap {}^\bullet v \neq \emptyset$, a contradiction.

Case 2: $E = [F * G * H]$. Proceed similarly as in Case 1.

Case 3: $E = F \| G$. Then (b) leads to a contradiction since it is obtained from proposition 53(2) that, without loss of generality, ${}^\bullet t \subseteq {}^\bullet\mathsf{box}(F)$, ${}^\bullet u \subseteq {}^\bullet\mathsf{box}(G)$, $t^\bullet \subseteq \mathsf{box}(F)^\bullet$ and $u^\bullet \subseteq \mathsf{box}(G)^\bullet$. Moreover, ${}^\bullet\Sigma \subseteq {}^\bullet t \cup {}^\bullet u$ and $\Sigma^\bullet \subseteq t^\bullet \cup u^\bullet$. Hence $t \in \mathsf{EX}_{\mathsf{box}(F)}$ and $u \in \mathsf{EX}_{\mathsf{box}(G)}$. But this means that ${}^\bullet v = v^\bullet = \emptyset$. If (a) holds then, clearly, $\alpha_i = \emptyset$, where i is such that w is the only transition of Σ_i. Moreover, $a \in \mathsf{max}_E$ **sy** $_{A \square \emptyset}$ can be shown easily using proposition 43(2). □

This section concludes with a useful result which states that the various sets and relations introduced in this section are preserved through duplication equivalence and synchronisation.

Proposition 66 Let A, B, C be synchronisation sets and E, F, G, H be boxes such that $E \simeq G$ and $F \simeq H$.

1. $\mathsf{set}_E = \mathsf{set}_G$ for $\mathsf{set} \in \{\mathsf{ex}, \mathsf{ccall}, \mathsf{ccint}, \mathsf{ccnoex}\}$.

2. $\mathsf{set}_E = \mathsf{set}_E \ \mathbf{sy}\ _A$ for $\mathsf{set} \in \{\mathsf{ccall}, \mathsf{ccint}\}$.

3. $\mathsf{set}^A_{EF} = \mathsf{set}^A_{GH}$ for $\mathsf{set} \in \{\mathsf{covall}, \mathsf{covmix}, \mathsf{covint}, \mathsf{covnoex}\}$.

4. $\mathsf{set}^A_{EF} = \mathsf{set}^A_E \ \mathbf{sy}\ _{B,F} \ \mathbf{sy}\ _C$ for $\mathsf{set} \in \{\mathsf{covall}, \mathsf{covmix}, \mathsf{covint}\}$.

5. $\mathsf{simex}_{EF} = \mathsf{simex}_{GH}$.

Proof: Follows from propositions 58, 60, 61 and 62. □

5.8 An axiomatisation of duplication equivalence

The axioms for duplication equivalence of box expressions are structured into six groups. Below, α stands for an arbitrary action, A and B for synchronisation sets, and a for a communication action.

Structural Identities. The first group of axioms (STR1-STR5) capture some basic structural identities. The axioms are sound not only with respect to duplication equivalence, but also with respect to net isomorphism. What they express is that the choice, parallel and sequential compositions are associative[5], and that the first two are also commutative operators.

Propagation of synchronisation. The first two of the next group of axioms (PROP1-PROP7) express simple (structural) facts about synchronisation, namely that applying synchronisation to a single action expression, or using the empty synchronisation set, has no effect at all. The third axiom allows one to collapse consecutive applications of the synchronisation operator. The remaining four axioms amount to saying that synchronisation propagates through the four composition operators.

[5]Therefore the parentheses in nested applications of sequence, choice and parallel composition operators may be omitted.

$$(E;F);G = E;(F;G) \qquad \text{STR1}$$

$$(E \,[]\, F) \,[]\, G = E \,[]\, (F \,[]\, G) \qquad \text{STR2}$$

$$E \,[]\, F = F \,[]\, E \qquad \text{STR3}$$

$$(E \| F) \| G = E \| (F \| G) \qquad \text{STR4}$$

$$E \| F = F \| E \qquad \text{STR5}$$

$$\alpha = \alpha \text{ sy } A \qquad \text{PROP1}$$

$$E = E \text{ sy } \emptyset \qquad \text{PROP2}$$

$$E \text{ sy } A \text{ sy } B = E \text{ sy } (A \cup B) \qquad \text{PROP3}$$

$$(E;F) \text{ sy } A = ((E \text{ sy } A);(F \text{ sy } A)) \text{ sy } A \qquad \text{PROP4}$$

$$(E \,[]\, F) \text{ sy } A = ((E \text{ sy } A) \,[]\, (F \text{ sy } A)) \text{ sy } A \qquad \text{PROP5}$$

$$(E \| F) \text{ sy } A = ((E \text{ sy } A) \| (F \text{ sy } A)) \text{ sy } A \qquad \text{PROP6}$$

$$[E * F * G] \text{ sy } A = [(E \text{ sy } A) * (F \text{ sy } A) * (G \text{ sy } A)] \text{ sy } A \qquad \text{PROP7}$$

$$\alpha \,[]\, \alpha = \alpha \qquad \text{DUPL}$$

Duplication. This group comprises only one axiom (DUPL). It captures the essence of duplication equivalence whereby a choice between two copies of the same action is ignored.

***ex*-actions.** The next axiom (EX) is used to deal with *ex*–actions as it allows these to be moved within a box expression. This is necessary, in particular, in order to make an expression with the main choice composition connective satisfy the first of the premises in the next axiom (LIFT1).

$$\frac{\alpha \in A \;\Rightarrow\; \forall B \in \mathsf{ccnoex}_{E\ \mathbf{sy}\ A} : \widehat{\alpha} \notin B}{(E\ \mathbf{sy}\ A) \,\Box\, \alpha = (E \,\Box\, \alpha)\ \mathbf{sy}\ A} \quad \text{EX}$$

$$\frac{A \in \mathsf{simex}_{E\ \mathbf{sy}\ \mathsf{a}, F\ \mathbf{sy}\ \mathsf{a}} \text{ and } a \in \mathsf{covint}^{A}_{EF} \cap \mathsf{covnoex}^{A}_{E\ \mathbf{sy}\ \mathsf{a}, F\ \mathbf{sy}\ \mathsf{a}}}{((E\ \mathbf{sy}\ \mathsf{a}) \,\Box\, (F\ \mathbf{sy}\ \mathsf{a}))\ \mathbf{sy}\ A = (E \,\Box\, F)\ \mathbf{sy}\ A\ \mathbf{sy}\ \mathsf{a}} \quad \text{LIFT1}$$

$$\frac{a \in \mathsf{covall}^{A}_{EF}}{((E\ \mathbf{sy}\ \mathsf{a}); (F\ \mathbf{sy}\ \mathsf{a}))\ \mathbf{sy}\ A = (E; F)\ \mathbf{sy}\ A\ \mathbf{sy}\ \mathsf{a}} \quad \text{LIFT2}$$

$$\frac{a \in \mathsf{covall}^{A}_{EF}}{((E\ \mathbf{sy}\ \mathsf{a}) \| (F\ \mathbf{sy}\ \mathsf{a}))\ \mathbf{sy}\ A = (E \| F)\ \mathbf{sy}\ A\ \mathbf{sy}\ \mathsf{a}} \quad \text{LIFT3}$$

$$\frac{a \in \mathsf{covall}^{A}_{EFG}}{[(E\ \mathbf{sy}\ \mathsf{a}) * (F\ \mathbf{sy}\ \mathsf{a}) * (G\ \mathbf{sy}\ \mathsf{a})]\ \mathbf{sy}\ A = [E * F * G]\ \mathbf{sy}\ A\ \mathbf{sy}\ \mathsf{a}} \quad \text{LIFT4}$$

$$(((E \,\Box\, a) \| (F \,\Box\, \widehat{a}))\ \mathbf{sy}\ \mathsf{a}) \,\Box\, \emptyset = ((E \,\Box\, a) \| (F \,\Box\, \widehat{a}))\ \mathbf{sy}\ \mathsf{a} \quad \text{INT1}$$

$$\frac{a \in \mathsf{covmix}^{A}_{EF} \cap \mathsf{ex}_E \cap \widehat{\mathsf{ex}_F}}{(((E\ \mathbf{sy}\ \mathsf{a}) \| (F\ \mathbf{sy}\ \mathsf{a}))\ \mathbf{sy}\ A) \,\Box\, \emptyset = (E \| F)\ \mathbf{sy}\ A\ \mathbf{sy}\ \mathsf{a}} \quad \text{INT2}$$

Lifting of synchronisation. The following four axioms (LIFT1-LIFT4) allow one to lift synchronisation sets to a higher level in the syntax tree of a box expression. The main application of these axioms is in the construction of maximal synchronisation sets.

Internal actions. The last axioms (INT1-INT2) capture two different ways in which a syntactically generated internal action can find its duplicate generated through synchronisation.

From now on only the box expressions which belong to Exp_0 are considered. Note that by applying any of the axioms to a box expression in Exp_0 one always produces an expression which also belongs to Exp_0. It will also be assumed that the set of communication actions $\mathcal{A}$ is finite.

5.9 Soundness of the axiom system

The first property of the axiom system that is established is soundness. Using the results obtained for nets and, in particular, the structural characterisation of the maximal synchronisation sets, it is fairly routine to show that the axiom system is indeed sound. If two box expressions, E and F, can be shown to be equivalent using these axioms, $E \equiv F$ will be written.

Theorem 6 For every box expression E in Exp_0, if $E \equiv F$ then $E \simeq F$.

Proof: As shown in [5], duplication equivalence is a congruence with respect to all the operators used in this paper. Hence it suffices to show that each of the axioms is sound. This is true for STR1-STR5 (since these are sound w.r.t. net isomorphism), PROP1, PROP2 and DUPL (which follows directly from the definition of $\Box$, **sy** and $\simeq$), PROP3 (using proposition 59(1)), PROP4-PROP7 (using proposition 49 and the property that synchronisation distributes over place multiplication and place addition), LIFT1-LIFT4 (using propositions 66(4, for covint) and 63, as well as axioms PROP4-PROP7) and EX

(using propositions 43(2) and 60), INT1 and INT2 (using proposition 43(2)).
□

Note that EX, DUPL and STR2 imply the soundness of two useful derived axioms, EX1 and EX2, where in the former $E \,\square\, \alpha$ can be equal to α.

$$((E \,\square\, \alpha)\ \mathbf{sy}\ A) \,\square\, \alpha \;=\; (E \,\square\, \alpha)\ \mathbf{sy}\ A \quad \text{EX1}$$

$$\frac{\alpha \notin A}{(E\ \mathbf{sy}\ A) \,\square\, \alpha = (E \,\square\, \alpha)\ \mathbf{sy}\ A} \quad \text{EX2}$$

5.10 Completeness of the axiom system

The proof of completeness is much more involved and is structured into two parts. The first one deals with maximal synchronisation sets showing that it is always possible to make the maximal synchronisation set of a box expression the outermost synchronisation, i.e., for every box expression E in Exp_0, $E \equiv E\ \mathbf{sy}\ \mathsf{max}_E$. The proof of this result relies heavily on the structural characterisation of the maximal synchronisation sets of box expressions. The second part begins with the development of a normal form for box expressions and ends up with the completeness proof.

Firstly, an auxiliary notion of context is introduced. A *context* is a term $\aleph$ derived from the following syntax:

$$\aleph ::= x\ \mathbf{sy}\ A \mid (\aleph \,\square\, \aleph)\ \mathbf{sy}\ A \mid (\aleph \| \aleph)\ \mathbf{sy}\ A \mid (\aleph; \aleph)\ \mathbf{sy}\ A \mid [\aleph * \aleph * \aleph]\ \mathbf{sy}\ A$$

where A is a synchronisation set and x is a place-holder (variable). It is assumed that within a given context, all the place holders are distinct. An

n-context is a context $\aleph$ with exactly n place holders. It can be denoted by $\aleph[x_1, \ldots, x_n]$, where $x_1, \ldots, x_n$ are the place-holders of $\aleph$ listed in the order in which they occur in $\aleph$. For every $i \leq n$ let $A^i_\aleph$ denote the synchronisation set such that $\mathbf{sy}\ A^i_\aleph$ was the synchronisation directly applied to the i-th place-holder, i.e., x_i. Moreover, for all $i, j \leq n$ let $A^{ij}_\aleph$ denote the union of all sets A such that x_i and x_j are in the scope an application of $\mathbf{sy}\ A$. For example, if

$$\aleph[x, y, z] = ((((x\ \mathbf{sy}\ \mathsf{a}) \| (y\ \mathbf{sy}\ \mathsf{c}))\ \mathbf{sy}\ \mathsf{d}) \| (z\ \mathbf{sy}\ \mathsf{e}))\ \mathbf{sy}\ \mathsf{f}$$

then $A^2_\aleph = \mathsf{c}$, $A^{22}_\aleph = \mathsf{c} \cup \mathsf{d} \cup \mathsf{f}$ and $A^{12}_\aleph = \mathsf{d} \cup \mathsf{f}$. $\aleph[p_1, \ldots, p_n]$ is used to denote the term resulting from replacing the place holders $x_1, \ldots, x_n$ by terms $p_1, \ldots, p_n$. Note that if each p_i is a box expression then so is $\aleph[p_1, \ldots, p_n]$. A context $\aleph$ is *saturated* (*reduced*) if, for every subcontext

$$(\ldots(\ldots)\ \mathbf{sy}\ A \ldots)\ \mathbf{sy}\ B$$

of $\aleph$, $B \subseteq A$ (resp. $B \cap A = \emptyset$).

A box expression E is in *standard form* if $E = \aleph[\alpha_1, \ldots, \alpha_n]$, for some n-context $\aleph$ and actions $\alpha_1, \ldots, \alpha_n$. Moreover, E is saturated (reduced) if $\aleph$ is saturated (resp. reduced). For a box expression E (or context $\aleph$), $\langle E \rangle$ (resp. $\langle \aleph \rangle$) is used to denote the term resulting from deleting all the occurrences of the synchronisation operator. Note that $\langle E \rangle$ is always a box expression. For example, $E = ((a\ \mathbf{sy}\ \mathsf{a}) \| (b\ \mathbf{sy}\ (\mathsf{a} \cup \mathsf{c})))\ \mathbf{sy}\ \mathsf{a}$ is a saturated box expression in standard form such that $\langle E \rangle = a \| b$.

It is always possible to transform an expression into one in standard form, saturated or reduced. In the first of the two results that follow, a suitable F can be obtained by applying axioms PROP2 and PROP3, and in the second one by applying axioms PROP4-PROP7.

Proposition 67 For every expression E there is an expression in standard form F such that $\langle E \rangle = \langle F \rangle$ and $E \equiv F$. □

Proposition 68 Let $E = \aleph[E_1, \ldots, E_n]$ be a box expression in standard form. Then there is a saturated (resp. reduced) n-context $\aleph'$ such that $E \equiv \aleph'[E_1, \ldots, E_n]$ and $\langle \aleph \rangle = \langle \aleph' \rangle$ and $A^{ij}_{\aleph} = A^{ij}_{\aleph'}$, for all $i, j \leq n$. □

For example, if $E = ((a \textbf{ sy } \mathsf{a}) \| (b \textbf{ sy } \mathsf{c})) \textbf{ sy } \mathsf{a}$ then $E \equiv ((a \textbf{ sy } \mathsf{a}) \| (b \textbf{ sy } (\mathsf{a} \cup \mathsf{c}))) \textbf{ sy } \mathsf{a}$ (exemplifying saturation) and $E \equiv ((a \textbf{ sy } \emptyset) \| (b \textbf{ sy } \mathsf{c})) \textbf{ sy } \mathsf{a}$ (exemplifying reduction).

5.10.1 Constructing maximal synchronisation sets

This section contains two results. The first, an auxiliary one, states that it is always possible to propagate an *ex*–action to the outside of a box expression.

Proposition 69 For every box expression E in Exp_0 and every $\alpha \in \mathsf{ex}_E$, $E \equiv E \,\square\, \alpha$.

Proof: By DUPL and STR2, it suffices to show that $E \equiv F \,\square\, \alpha$, for some expression F. Induction on the structure of $\langle E \rangle$ is used. By proposition 67, it may be assumed that E is in standard form. The base case is $E = \alpha \textbf{ sy } A$. Then $\mathsf{ex}_E = \{\alpha\}$ and, by EX1, $\alpha \textbf{ sy } A \equiv (\alpha \textbf{ sy } A) \,\square\, \alpha$. The induction step is split into three cases.

Case 1: $E = (F; G) \textbf{ sy } A$ or $E = [F * G * H] \textbf{ sy } A$. Then $\mathsf{ex}_E = \emptyset$ and there is nothing to prove.

Case 2: $E = (F \| G) \textbf{ sy } A$. Then, by the definition of ex, $\alpha = \emptyset$ and there is $a \in A$ such that $a \in \mathsf{ex}_F$ and $\hat{a} \in \mathsf{ex}_G$. By the induction hypothesis, $F \equiv F \,\square\, a$ and $G \equiv G \,\square\, \hat{a}$. Thus, by INT1, PROP3 and EX2,

$$\begin{aligned} E &\equiv ((F \,\square\, a) \| (G \,\square\, \hat{a})) \textbf{ sy } \mathsf{a} \textbf{ sy } A \\ &\equiv (((F \,\square\, a) \| (G \,\square\, \hat{a})) \textbf{ sy } \mathsf{a} \,\square\, \emptyset) \textbf{ sy } A \\ &\equiv ((F \,\square\, a) \| (G \,\square\, \hat{a})) \textbf{ sy } \mathsf{a} \textbf{ sy } A \,\square\, \emptyset. \end{aligned}$$

Case 3: $E = (F \,\square\, G) \textbf{ sy } A$. Then, by proposition 68, there is a saturated n-context $\aleph$, $n \geq 2$, and box expressions $E_1, \ldots, E_n$ such that $E \equiv \aleph[E_1, \ldots, E_n]$,

$\aleph$ uses only choice and synchronisation, and, for every $i \leq n$, the topmost operator of E_i (if E_i is not a single action expression) is neither choice nor synchronisation. It may now be observed that from $A^i_\aleph = A^{ii}_\aleph$ for every $i \leq n$ (as $\aleph$ is saturated) and the definition of ex, it follows that there is $k \leq n$ such that $\alpha \in \mathsf{ex}_{E_k}$ **sy** $_{A^k_\aleph}$. Hence, by the induction hypothesis, E_k **sy** $A^k_\aleph \equiv (E_k$ **sy** $A^k_\aleph) \,\Box\, \alpha$. Then, by repeatedly applying STR2, STR3 and EX1 , one can show that $E \equiv H \,\Box\, \alpha$, for some H. □

The next result shows that it is always possible to make the maximal synchronisation set of an expression the outermost synchronisation.

Proposition 70 For every expression E in Exp_0, $E \equiv E$ **sy** max_E.

Proof: By PROP3 and the finiteness of $\mathcal{A}$, it suffices to show that for every expression E and action $a \in \mathsf{max}_E$, $E \equiv F$ **sy** a for some expression F. The proof is by induction on the structure of $\langle E \rangle$. By proposition 67, it may be assumed that E is in standard form (note that applying proposition 67 does not change the expression if the occurrences of the synchronisation operator are disregarded; hence as far as the inductive proof on the structure of $\langle E \rangle$ is concerned, applying the proposition is harmless). The base case is $E = \alpha$ **sy** A. Then, using PROP1 and PROP3, the following is obtained

$$\alpha \textbf{ sy } A \equiv \alpha \textbf{ sy } \mathcal{A} \textbf{ sy } A \equiv \alpha \textbf{ sy } \mathcal{A} \equiv \alpha \textbf{ sy } \mathcal{A} \textbf{ sy } \mathsf{a}.$$

In the induction step, if $E = (F;G)$ **sy** A then, by proposition 63, $a \in \mathsf{max}_F$ **sy** $_A \cap \mathsf{max}_G$ **sy** $_A$ and $\mathsf{a} \in \mathsf{coval}^A_{FG}$. By the induction hypothesis, $F \equiv F$ **sy** a and $G \equiv G$ **sy** a. Hence, by LIFT2,

$$E \equiv ((F \textbf{ sy } \mathsf{a}); (G \textbf{ sy } \mathsf{a})) \textbf{ sy } A \equiv (F;G) \textbf{ sy } A \textbf{ sy } \mathsf{a}.$$

If $E = (F \| G)$ **sy** A or $E = [E * F * G]$ **sy** A then the proof follows in a similar way.

The only complicated case to consider is that of $E = (F \,\Box\, G)$ **sy** A. Then there is an n-context $\aleph'$, $n \geq 2$, and box expressions $E_1, \ldots, E_n$ such that

$E = \aleph'[E_1, \ldots, E_n]$, $\aleph'$ uses only choice and synchronisation, and, for every $i \leq n$, the topmost operator of every E_i (if E_i is not a single action expression) is different from choice and synchronisation.

For every $i \leq n$, let $F_i = E_i \,\square\, a_1 \,\square\, \ldots \,\square\, a_m$ where $\{a_1, \ldots, a_m\} = \{E_j \mid E_j \in A^{ij}_{\aleph} \wedge j \neq i\}$ (if $m = 0$ then $F_i = E_i$). A three-stage transformation of E into an expression whose structure will allow us to apply proposition 63 is performed. It may be assumed, by proposition 68, that E is reduced.

In the first stage, by applying STR2, STR3, DUPL and EX2, $E \equiv F$ where $F = \aleph'[F_1, \ldots, F_n]$. In the second stage, if $\emptyset \in \mathsf{ex}_E$ then, using proposition 69 and axioms STR2, STR3, DUPL and EX2, F may be transformed into an equivalent expression $H = \aleph'[H_1, \ldots, H_n]$ where $H_i = F_i \,\square\, \emptyset$, for every $i \leq n$. In the third stage, using proposition 68, H may be transformed into an equivalent expression $J = \aleph[H_1, \ldots, H_n]$ such that $\langle \aleph \rangle = \langle \aleph' \rangle$, $\aleph$ is saturated and $A^{ij}_{\aleph} = A^{ij}_{\aleph'}$, for all $i, j \leq n$.

It may now be observed that for every subcontext

$$(\aleph_1[x_k, \ldots, x_l] \,\square\, \aleph_2[x_{l+1}, \ldots, x_m]) \;\mathbf{sy}\; A^{km}_{\aleph}$$

of $\aleph$, the expression $(\aleph_1[H_k, \ldots, H_l] \,\square\, \aleph_2[H_{l+1}, \ldots, H_m])$ **sy** $A^{km}_{\aleph}$ does satisfy the pre-condition in the second part of proposition 63. In particular, this and the fact that $\aleph$ is saturated imply that, for every $i \leq n$, $a \in \mathsf{max}_{H_i}\ \mathbf{sy}\ A^i_{\aleph}$.

It is now shown that for every $i \leq n$, H_i **sy** $A^i_{\aleph} \equiv K$ **sy a**, for some K. Firstly it is observed that if $a \in A^{ii}_{\aleph} = A^i_{\aleph}$ then by PROP3 nothing further is required. Moreover, if it is the case that $H_i = E_i$ then the induction hypothesis may be used. So suppose that $a \notin A^i_{\aleph}$ and $H_i = (E_i \,\square\, \psi)$. Note that in such a case, by construction, neither a nor $\widehat{a}$ occurs in ψ. Four cases are considered:

Case 1: $E_i = \alpha$. Then $H_i \equiv H_i$ **sy a** which follows from PROP1, STR2, STR3, EX2 and the fact that ψ does not contain a nor $\widehat{a}$. Hence, by PROP3, H_i **sy** $A^i_{\aleph} \equiv H_i$ **sy** $A^i_{\aleph}$ **sy a**.

Case 2: $E_i = F;G$. Then, by proposition 65(1), $a \in \mathsf{max}_{E_i}\ \mathbf{sy}\ A^i_{\aleph}$ and, since $\langle E_i \rangle$ is a proper subexpression of $\langle E \rangle$, by the induction hypothesis and

axioms PROP3 and PROP5,

$$\begin{aligned}(E_i \,[]\, \psi)\ \mathbf{sy}\ A^i_{\aleph} &\equiv ((E_i\ \mathbf{sy}\ A^i_{\aleph})\,[]\,(\psi\ \mathbf{sy}\ A^i_{\aleph}))\ \mathbf{sy}\ A^i_{\aleph}\\ &\equiv ((E_i\ \mathbf{sy}\ \mathsf{a}\ \mathbf{sy}\ A^i_{\aleph})\,[]\,(\psi\ \mathbf{sy}\ A^i_{\aleph}))\ \mathbf{sy}\ A^i_{\aleph}\\ &\equiv ((E_i\ \mathbf{sy}\ \mathsf{a})\,[]\,\psi)\ \mathbf{sy}\ A^i_{\aleph}.\end{aligned}$$

It then follows that since ψ contains neither a nor $\widehat{a}$, by PROP3 and EX2, the last expression may be rewritten to $(E_i \,[]\, \psi)\ \mathbf{sy}\ A^i_{\aleph}\ \mathbf{sy}\ \mathsf{a}$.

Case 3: $E_i = [F * G * H]$. Similar to Case 2.

Case 4: $E_i = F\|G$. If $a \in \mathsf{max}_{E_i}\ \mathbf{sy}\ _{A^i_{\aleph}}$ then proceed in a similar way to Case 2. Otherwise, by propositions 64 and 65(2):

$$a \in \mathsf{max}_F\ \mathbf{sy}\ _{A^i_{\aleph}} \cap \mathsf{max}_G\ \mathbf{sy}\ _{A^i_{\aleph}} \cap \mathsf{covmix}^{A^i_{\aleph}}_{FG}$$

and (without loss of generality and by applying STR2 and STR3, if necessary) $a \in \mathsf{ex}_F$, $\widehat{a} \in \mathsf{ex}_G$ and $\psi = \emptyset \,[]\, \phi$. By the induction hypothesis and PROP3, $F\ \mathbf{sy}\ A^i_{\aleph} \equiv F\ \mathbf{sy}\ \mathsf{a}\ \mathbf{sy}\ A^i_{\aleph}$ and $G\ \mathbf{sy}\ A^i_{\aleph} \equiv G\ \mathbf{sy}\ \mathsf{a}\ \mathbf{sy}\ A^i_{\aleph}$. Hence, by PROP6 and INT2,

$$\begin{aligned}(F\|G)\ \mathbf{sy}\ A^i_{\aleph} \,[]\, \emptyset &\equiv ((F\ \mathbf{sy}\ A^i_{\aleph})\|(G\ \mathbf{sy}\ A^i_{\aleph}))\ \mathbf{sy}\ A^i_{\aleph} \,[]\, \emptyset\\ &\equiv ((F\ \mathbf{sy}\ \mathsf{a})\|(G\ \mathbf{sy}\ \mathsf{a}))\ \mathbf{sy}\ A^i_{\aleph} \,[]\, \emptyset\\ &\equiv (F\|G)\ \mathbf{sy}\ A^i_{\aleph}\ \mathbf{sy}\ \mathsf{a}.\end{aligned}$$

Then proceed as in Case 2 to show that $H_i\ \mathbf{sy}\ A^i_{\aleph} \equiv ((F\|G) \,[]\, \phi)\ \mathbf{sy}\ A^i_{\aleph}\ \mathbf{sy}\ \mathsf{a}$.

Hence, it has been shown that for every $i \leq n$, $H_i\ \mathbf{sy}\ A^i_{\aleph} \equiv K\ \mathbf{sy}\ \mathsf{a}$, for some K. Thus, by PROP3, $H_i\ \mathbf{sy}\ A^i_{\aleph} \equiv H_i\ \mathbf{sy}\ A^i_{\aleph}\ \mathbf{sy}\ \mathsf{a}$, for every $i \leq n$. The last part of the proof is carried out assuming that $n = 2$ (the argument extends easily to $n > 2$).

Assume $J = ((H_1\ \mathbf{sy}\ A_1) \,[]\, (H_2\ \mathbf{sy}\ A_2))\ \mathbf{sy}\ B$. It has been shown that $H_i\ \mathbf{sy}\ A_i \equiv H_i\ \mathbf{sy}\ A_i\ \mathbf{sy}\ \mathsf{a}$, for $i = 1, 2$. Moreover, as has already been observed, $B \in \mathsf{simex}_{H_1}\ \mathbf{sy}\ _{A_1, H_2}\ \mathbf{sy}\ _{A_2}$. Hence, by proposition 63,

$$a \in \mathsf{covint}^B_{H_1}\ \mathbf{sy}\ _{A_1, H_2}\ \mathbf{sy}\ _{A_2} \cap \mathsf{covnoex}^B_{H_1}\ \mathbf{sy}\ _{A_1, H_2}\ \mathbf{sy}\ _{A_2}.$$

Also, by proposition 66(3,5):

$$\begin{aligned} B &\in \mathsf{simex}_{H_1}\ \mathbf{sy}\ A_1\ \mathbf{sy}\ \mathsf{a}, H_2\ \mathbf{sy}\ A_2\ \mathbf{sy}\ \mathsf{a} \\ a &\in \mathsf{covnoex}^{B}_{H_1}\ \mathbf{sy}\ A_1\ \mathbf{sy}\ \mathsf{a}, H_2\ \mathbf{sy}\ A_2\ \mathbf{sy}\ \mathsf{a}. \end{aligned}$$

Hence LIFT1, may be applied in the following way

$$\begin{aligned} J &\equiv ((H_1\ \mathbf{sy}\ A_1\ \mathbf{sy}\ \mathsf{a}) \,\square\, (H_2\ \mathbf{sy}\ A_2\ \mathbf{sy}\ \mathsf{a}))\ \mathbf{sy}\ B \\ &\equiv ((H_1\ \mathbf{sy}\ A_1) \,\square\, (H_2\ \mathbf{sy}\ A_2))\ \mathbf{sy}\ B\ \mathbf{sy}\ \mathsf{a} \end{aligned}$$

which completes the proof. □

5.10.2 De-synchronisation

An auxiliary operator on nets is now introduced. This can be thought of as an 'inverse' of the synchronisation operator, or *de-synchronisation.* For a labelled net Σ and a synchronisation set A, let $\Sigma\ \mathsf{unsy}\ A$ denote the net obtained from Σ by deleting all the $\emptyset$–labelled transitions t for which there are A–synchronisable transitions u and w such that $t \bowtie_\Sigma uw$.

Proposition 71 Let Σ_1 and Σ_2 be labelled nets, and A and B be synchronisation sets.

1. $\Sigma_1\ \mathsf{unsy}\ \emptyset = \Sigma_1$.
2. If $\Sigma_1 \simeq \Sigma_2$ then $\Sigma_1\ \mathsf{unsy}\ A \simeq \Sigma_2\ \mathsf{unsy}\ A$.
3. If $A \subseteq \mathsf{max}_{\Sigma_1}$ then $\Sigma_1 \simeq (\Sigma_1\ \mathsf{unsy}\ A)\ \mathbf{sy}\ A$.
4. If $A \subseteq B$ then $(\Sigma_1\ \mathbf{sy}\ A)\ \mathsf{unsy}\ B = \Sigma_1\ \mathsf{unsy}\ B$.
5. If Σ_1 is a pre-box then $\Sigma_1\ \mathsf{unsy}\ A$ is also a pre-box.
6. The $\mathsf{unsy}\ A$ operator distributes over the sequence, parallel and iteration composition.

Proof: (1), (4) and (5) follow directly from the definitions of **sy** and unsy.

(2) Suppose that $\Sigma_1 \simeq_h \Sigma_2$. Observe that if $t \simeq_h u$ then $t \in T_{\Sigma_1 \,\mathsf{unsy}\, A}$ if and only if $u \in T_{\Sigma_2 \,\mathsf{unsy}\, A}$ which follows directly from proposition 42(1). Moreover, it is easy to see that for all nodes n and n' in $\Sigma_1 \,\mathsf{unsy}\, A$, $n \bowtie_{\Sigma_1} n'$ if and only if $n \bowtie_{\Sigma_1 \,\mathsf{unsy}\, A} n'$. And, similarly, for all nodes m and m' in $\Sigma_2 \,\mathsf{unsy}\, A$, $m \bowtie_{\Sigma_2} m'$ if and only if $m \bowtie_{\Sigma_2 \,\mathsf{unsy}\, A} m'$. One can then show that

$$g = \{([n]_{\simeq}^{\Sigma_1 \,\mathsf{unsy}\, A}, [m]_{\simeq}^{\Sigma_2 \,\mathsf{unsy}\, A}) \mid n \in S_{\Sigma_1} \cup T_{\Sigma_1 \,\mathsf{unsy}\, A} \wedge n \simeq_h m\}$$

is an isomorphism such that $\Sigma_1 \,\mathsf{unsy}\, A \simeq_g \Sigma_2 \,\mathsf{unsy}\, A$.

(3) Since $\Sigma_1 \simeq \Sigma_1$ **sy** A, it suffices to show that Σ_1 **sy** $A \simeq (\Sigma_1 \,\mathsf{unsy}\, A)$ **sy** A. The latter follows directly from proposition 43(2) and the definitions of **sy** and unsy (essentially, Σ_1 **sy** A is $(\Sigma_1 \,\mathsf{unsy}\, A)$ **sy** A with possibly duplicates of some $\emptyset$–labelled transitions added).

(6) It suffices to observe that, by proposition 41, if Σ_2 is any of the nets (Σ_i or Σ_i') used in the definition of sequence, concurrent or iteration composition Σ, and t is a transition in Σ_2, then there are no transitions, u and w, of which at least one belongs to the remaining nets forming Σ, such that $t \bowtie_{\Sigma} uw$. □

De-synchronising a box does not necessarily yield a box. For example, if $E = (((a \,\Box\, b); b) \| (\widehat{a} \,\Box\, \widehat{b}))$ **sy** b then $\mathsf{box}(E) \,\mathsf{unsy}\, \mathsf{a}$ is the net in Figure 5.10 which, as one can easily see, is neither a box nor is duplication equivalent to any box. But what can be said about a de-synchronised box is that it is a box with some transitions added in a way which resembles 'local' synchronisation.

Proposition 72 Let Σ', $\Sigma = [\Sigma' \,\mathsf{unsy}\, A]_{\simeq}$, be boxes, and A be a synchronisation set. Then there is a box Σ_a generated by a synchronisation-free expression and a set of $\emptyset$-labelled transitions T of Σ such that Σ_a is isomorphic to Σ with the transitions T deleted. Moreover, for every transition $t \in T$ there are transitions $u, w \in T_{\Sigma} - T$ such that $t \bowtie_{\Sigma} uw$.

Proof: Let $\Sigma' = \mathsf{box}(E)$. Observe that if the result is shown for the modified box obtained by omitting all the transitions which resulted from synchronisations, then the proposition will also hold for the original box. In other words, it suffices to show the result assuming that E does not contain any application of the synchronisation operator. Then, one can show that Σ is isomorphic to $\mathsf{box}(F)$, where F is the expression obtained from E by removing, from each subexpression $G = G_1 \,[]\, \ldots \,[]\, G_k$ all the expressions $G_i = \alpha$ such that there is $j < i$ satisfying $G_j = G_i$ and, moreover, if $\mathsf{potex}_G \cap A \neq \emptyset$ then also all the expressions of the form $G_i = \emptyset$ (the proof can be carried out by induction on the structure of E and using proposition 71(6)). □

An *ex-path* of a labelled net Σ is a sequence of nodes $\pi = n_0 \ldots n_k$ such n_0 is an entry place, n_k is an exit place, and $n_{i-1} \in {}^\bullet n_i$, for every $i \leq k$. Σ is *ex-connected*, if for every place s in Σ there is an *ex–path* to which s belongs. The following is an easy corollary of the last result and the fact proved in Chapter 3 that boxes generated by synchronisation-free expressions are *ex*-connected.

Corollary 9 If Σ is a box and A is a synchronisation set then $\Sigma\,\mathsf{unsy}\,A$ is *ex*-connected. □

De-synchronisation distributes over the sequence, parallel and iteration composition. However, this does not extend to the choice operator. For example, if $\Sigma_a = \mathsf{box}(a\|\widehat{a})$ and $\Sigma_b = \mathsf{box}(b\|\widehat{b})$ then

$$\begin{aligned}((\Sigma_a\ \mathsf{sy}\ \mathsf{a}) \,[]\, (\Sigma_b\ \mathsf{sy}\ \mathsf{b}))\,\mathsf{unsy\,a} &= \Sigma_a \,[]\, \Sigma_b \not\equiv \Sigma_a \,[]\, (\Sigma_b\ \mathsf{sy}\ \mathsf{b}) \\ &= ((\Sigma_a\ \mathsf{sy}\ \mathsf{a})\,\mathsf{unsy\,a}) \,[]\, ((\Sigma_b\ \mathsf{sy}\ \mathsf{b})\,\mathsf{unsy\,a}).\end{aligned}$$

An important case when de-synchronisation distributes over choice is provided by the next result.

Proposition 73 Let $\Sigma_1, \ldots, \Sigma_2$ and $\Sigma = \Sigma_1 \,[]\, \ldots \,[]\, \Sigma_k$ be boxes in Box_0 and A be a synchronisation set such that, for every $i \leq k$, $A \subseteq \mathsf{max}_{\Sigma_i}$ and if $\mathsf{ex}_{\Sigma_i} \neq \emptyset$ then $\Sigma_i = \mathsf{box}(\alpha)$ for some action α. Then

$$\Sigma\,\mathsf{unsy}\,A = (\Sigma_1\,\mathsf{unsy}\,A) \,[]\, \ldots \,[]\, (\Sigma_k\,\mathsf{unsy}\,A).$$

Proof: To show the result it suffices to prove that if $t \in T_{\Sigma_i}$ and there are A-synchronisable transitions u and w in Σ satisfying $t \bowtie_\Sigma uw$, then $u, w \in T_{\Sigma_i}$. Two cases are considered.

Case 1: $\mathsf{ex}_{\Sigma_i} = \emptyset$. If $u, w \notin T_{\Sigma_i}$ then a contradiction is obtained with propositions 41 and 54(2) and $t \notin \mathsf{EX}_\Sigma$ and $\Sigma_i \in \mathsf{Box}_0$. If $u \in T_{\Sigma_i}$ and $w \notin T_{\Sigma_i}$ then, by proposition 41, $w \in \mathsf{EX}_\Sigma$. Hence $t \bowtie_{\Sigma_i} u\delta$ and from proposition 54(1) it follows that $\mathsf{EX}_{\Sigma_i} \neq \emptyset$, a contradiction. Thus $u, w \in T_{\Sigma_i}$.

Case 2: $\mathsf{ex}_{\Sigma_i} \neq \emptyset$. Then $\Sigma_i = \mathsf{box}(\emptyset)$ and, using proposition 41, one can easily see that both u and w belong to the same Σ_j. Hence, by $A \subseteq \mathsf{max}_{\Sigma_j}$ and $uw \bowtie_{\Sigma_j} \delta$, $\mathsf{ex}_{\Sigma_j} \neq \emptyset$, a contradiction. □

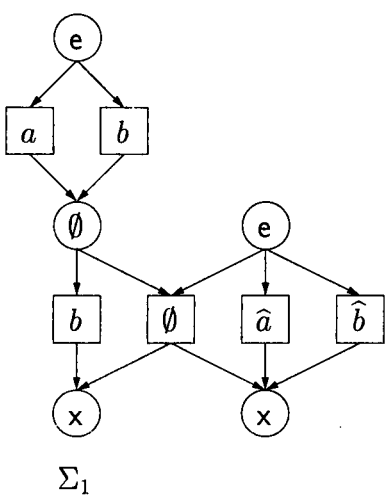


Figure 5.10: De-synchronised box may not be a box.

The next definition and the proposition that follows deal with the problem of a unique representation of a net as a composition of other, smaller, nets. Below, a pre-box Σ is *choice-decomposable* (*sequence-decomposable*) if there are pre-boxes Σ_a and Σ_b such that $\Sigma \simeq \Sigma_a \,[]\, \Sigma_b$ (resp. $\Sigma \simeq \Sigma_a ; \Sigma_b$). Note that $\mathsf{box}(\alpha)$ is choice-decomposable, for every action α. An *iteration / parallel / sequence / choice decomposition* of a pre-box Σ is a sequence of pre-boxes $\Sigma_1, \ldots, \Sigma_k$ such that, respectively, the following hold:

- $k = 3$ and $\Sigma \simeq [\Sigma_1 * \Sigma_2 * \Sigma_3]$.

- $k \geq 2$ and $\Sigma \simeq \Sigma_1 \| \ldots \| \Sigma_k$ and, for every $i \leq k$, Σ_i is connected.
- $k \geq 2$ and $\Sigma \simeq \Sigma_1; \ldots; \Sigma_k$ and, for every $i \leq k$, Σ_i is not sequence-decomposable.
- $k \geq 2$ and $\Sigma \simeq \Sigma_1 \,\square \ldots \square\, \Sigma_k$ and, for every $i \leq k$, if Σ_i is choice-decomposable then $\Sigma_i \simeq \mathsf{box}(\alpha)$ for some action α and $\Sigma_i \not\simeq \Sigma_j$ for all $j \neq i$.

Proposition 74 Let Σ be a box and A be a synchronisation set.

1. $\Sigma\,\mathsf{unsy}\,A$ has at most one (up to duplication equivalence and permutation) parallel decomposition and choice decomposition.
2. $\Sigma\,\mathsf{unsy}\,A$ has at most one (up to duplication equivalence) sequence decomposition and iteration decomposition.

Proof: This proposition is proved using the results obtained in Chapter 3. A sketch of how this can be done for the case of sequence decomposition is give below.

To start with, one can show that if $\Sigma'_1, \ldots, \Sigma'_k$ is a sequence decomposition of a pre-box Σ' then $[\Sigma'_1]_\simeq, \ldots, [\Sigma'_k]_\simeq$ is a sequence decomposition of $[\Sigma']_\simeq$ such that $[\Sigma']_\simeq = [\Sigma'_1]_\simeq; \ldots; [\Sigma'_k]_\simeq$. Hence it suffices to show that $\Sigma_a = [\Sigma\,\mathsf{unsy}\,A]_\simeq$ has at most one sequence decomposition $\Sigma_1, \ldots, \Sigma_k$ such that $\Sigma_a = \Sigma_1; \ldots; \Sigma_k$.

By proposition 72, it may be assumed that there is a box Σ_b generated by a synchronisation-free expression E (for which the results obtained in Chapter 3 can be applied) and transitions t, u, w in Σ_a such that $t \bowtie_{\Sigma_a} uw$, $(u, w) \in \mathsf{syn}_A$ and Σ_b is Σ_a with t deleted (i.e. it is assumed that $|T| = 1$ where T is as in the formulation of proposition 72; the argument is similar for $|T| > 1$). Then the following hold:

(i) Every representation of Σ_a as a sequential composition of pre-boxes induces a corresponding representation of Σ_b. More precisely, if $\Sigma_a =$

$\Sigma_1;\ldots;\Sigma_k$, for some pre-boxes Σ_i, and $t \in T_{\Sigma_l}$ then $\Sigma_b = \Sigma_{b_1};\ldots;\Sigma_{b_k}$ where Σ_{b_l} is Σ_l with t deleted, and $\Sigma_{b_i} = \Sigma_i$, for all $i \neq l$.

(ii) It was shown in Chapter 3 that there is a unique sequence of boxes $\Sigma_{f_1},\ldots,\Sigma_{f_m}$ for Σ_b such that whenever $\Sigma_b = \Sigma_{b_1};\ldots;\Sigma_{b_l}$ there are (unique) integers $0 = m_0 < m_1 < \ldots < m_l = m$ such that, for every $i \leq l$, $\Sigma_{b_i} = \Sigma_{f_{1+m_{i-1}}};\ldots;\Sigma_{f_{m_i}}$ (in particular, this means that no F_i is sequence decomposable). This and (i) implies that for Σ_a there is at most one sequence of non-sequence-decomposable pre-boxes $\Sigma_1,\ldots,\Sigma_k$ such that $\Sigma_a = \Sigma_1;\ldots;\Sigma_k$ (basically, if $u \in T_{\Sigma_{f_x}}$ and $w \in T_{\Sigma_{f_y}}$ $(x \leq y)$ then $\Sigma_1,\ldots,\Sigma_k$ is
$\Sigma_{f_1},\ldots,\Sigma_{f_{x-1}},(\Sigma_{f_x};\Sigma_{f_{x+1}};\ldots;\Sigma_{f_y}),\Sigma_{f_{y+1}},\ldots,\Sigma_{f_m})$. □

5.10.3 Normal form box expressions

A box expression E in Exp_0 is in *normal form* if it is in one of the following five types. Below, $A = \mathsf{max}_E$ and each E_i is an expression in normal form such that $A \subseteq \mathsf{max}_{E_i}$. Moreover, Σ denotes $\mathsf{box}(E)\,\mathsf{unsy}\,A$ and Σ_i denotes $\mathsf{box}(E_i)\,\mathsf{unsy}\,A$.

- Type-a $E = \alpha$ for some action α.
- Type-i $E = [E_1 * E_2 * E_3]$ **sy** A.
- Type-p $E = (E_1 \| \ldots \| E_k)$ **sy** A and $\Sigma_1,\ldots,\Sigma_k$ is a parallel decomposition of Σ.
- Type-c $E = (E_1 \,\square\, \ldots \,\square\, E_k)$ **sy** A and $\Sigma_1,\ldots,\Sigma_k$ is a choice decomposition of Σ.
- Type-s $E = (E_1;\ldots;E_k)$ **sy** A and $\Sigma_1,\ldots,\Sigma_k$ is a sequence decomposition of Σ.

The next task is to show that duplication equivalent expressions in normal form are equal (up to permutation of subexpressions in choice and parallel composition contexts). The first step is to show that any two duplication equivalent expressions are of the same type. The proof will rely on the notions of internal connectedness and internal interface, introduced in Chapter 2. Note that the application of the results based on internal connectedness and internal interfaces from Chapter 3 is possible due to corollary 9. The relevant proofs from Chapter 3 can easily be adapted and therefore their detailed exposition is omitted here, referring instead to the appropriate parts of Chapter 3.

A pre-box Σ is *internally connected* if it is connected after removing all the entry and exit places. It has an *internal interface* if there is a set of internal places P such that if P is deleted then Σ can be divided into two disjoint subgraphs with the nodes N_1 and N_2 such that: (i) each node in N_1 is connected to an entry place and not connected to any exit place; (ii) each node in N_2 is connected to an exit place and not connected to any entry place; and (iii) if Σ_i is taken to be Σ with the nodes N_i deleted $(i = 1, 2)$, then P is a $\otimes$-set for Σ_1 and Σ_2. For example, $P = \{s_2, s_3\}$ is the only internal interface of the net in Figure 5.5.

Proposition 75 Let E be an expression in normal form and m be the number of transitions in $\Sigma = \mathsf{box}(E)\ \mathsf{unsy}\ \mathsf{max}_E$.

1. If E is of type-a then $m = 1$.
2. If E is of type-p then $m > 1$ and Σ is not connected.
3. If E is of type-c then $m > 1$ and Σ is connected and not internally connected.
4. If E is of type-i then $m > 1$ and Σ is connected and internally connected and has no internal interface.
5. If E is of type-s then $m > 1$ and Σ is connected and internally connected and has at least one internal interface.

Proof: Let the Σ's and k be as in the definition of normal form. Note that, by proposition 71(5) and corollary 9, each Σ_i is an ex-connected pre-box.

(1) obviously holds as well as $m > 1$ in (2)–(5).

(2) Then $\Sigma \simeq \Sigma_1 \| \ldots \| \Sigma_k$. This and $k \geq 2$ means that Σ is not connected.

(3) Then $\Sigma \simeq \Sigma_1 \,[]\, \ldots \,[]\, \Sigma_k$. Since $k \geq 2$ and each Σ_i is ex-connected, it follows that Σ is connected but not internally connected (c.f. Proposition 5 in Chapter 3).

(4) Then $\Sigma \simeq [\Sigma_1 * \Sigma_2 * \Sigma_3]$. Since each Σ_i is ex-connected, it follows that Σ is connected, internally connected, and has no internal interface (c.f. Propositions 5 and 7 in Chapter 3).

(5) Then $\Sigma \simeq \Sigma_1; \ldots; \Sigma_k$. Since $k \geq 2$ and each Σ_i is ex-connected, it follows that Σ is connected, internally connected, and has an internal interface (c.f. Propositions 5 and 7 in Chapter 3). □

Directly from propositions 71(2) and 75, and the fact that connectedness, internal connectedness and having an internal interface are all net properties preserved by duplication equivalence, the following result is obtained.

Corollary 10 Duplication equivalent normal form expressions are of the same type. □

Moreover, one can show that duplication equivalent normal form expressions are equal.

Proposition 76 If E and F are duplication equivalent expressions in normal form then $E = F$ up to permutation of the components in subexpressions of the form $E_1 \| \ldots \| E_k$ and $E_1 \,[]\, \ldots \,[]\, E_k$.

Proof: Clearly, $\mathsf{max}_E = \mathsf{max}_F = A$ and, by corollary 10, both E and F are of the same type. The proof proceeds by induction on the structure of E. The base case is $E = \alpha$ and $F = \alpha'$ which clearly implies $E = F$.

In the inductive step, suppose that E and F are of type-p and, moreover, that $E = (E_1 \| \ldots \| E_k)$ sy A and $E = (F_1 \| \ldots \| F_m)$ sy A. By proposition 71(2), $\mathsf{box}(E)$ unsy $A \simeq \mathsf{box}(F)$ unsy A. Hence, by proposition 74(1), $k = m$ and, without loss of generality (one can always use STR4 and STR5), $\mathsf{box}(E_i)$ unsy $A \simeq \mathsf{box}(F_i)$ unsy A and so

$$\mathsf{box}(E_i) \text{ unsy } A \text{ sy } A \simeq \mathsf{box}(F_i) \text{ unsy } A \text{ sy } A,$$

for every $i \leq k$. Thus, by $A \subseteq \mathsf{max}_{E_i} = \mathsf{max}_{F_i}$ and proposition 71(3), $E_i \simeq F_i$, for every $i \leq k$. Now, since both E_i and F_i are in normal form, by the induction hypothesis, $E_i = F_i$ (up to permutation of subexpressions in choice and parallel contexts). Hence the proposition holds. The proofs for E of type-c, type-i and type-s are similar. □

The next proposition is an auxiliary result used later, in proposition 78, to transform an expression into a normal form expression.

Proposition 77 Let E be a normal form expression and $A \subseteq \mathsf{max}_E$ be a synchronisation set.

1. If E is of type-p and $\mathsf{box}(E)$ unsy A is not connected then there is $F = (F_1 \| \ldots \| F_k)$ sy A $(k \geq 2)$ such that $E \equiv F$ and, for every $i \leq k$, $A \subseteq \mathsf{max}_{F_i}$ and $\mathsf{box}(F_i)$ unsy A is connected.

2. If E is of type-s and $\mathsf{box}(E)$ unsy A is choice decomposable then there is an expression $F = (F_1; \ldots; F_k)$ sy A $(k \geq 2)$ such that $E \equiv F$ and, for every $i \leq k$, $A \subseteq \mathsf{max}_{F_i}$ and $\mathsf{box}(F_i)$ unsy A is not sequence-decomposable.

3. If E is of type-c, $\mathsf{ex}_E = \emptyset$ and $\mathsf{box}(E)$ unsy A is choice decomposable then there is an expression $F = (F_1 \,[]\, \ldots \,[]\, F_k)$ sy A $(k \geq 2)$ such that $E \equiv F$ and, for every $i \leq k$, $A \subseteq \mathsf{max}_{F_i}$ and $\mathsf{box}(F_i)$ unsy A is not choice-decomposable.

Proof: (1) Suppose $E = (E_1 \| \ldots \| E_m)$ **sy** B. Define a relation ρ on $\{1, \ldots, m\}$ in such a way that $(i,j) \in \rho$ if $i \neq j$ and

$$(E_i \| E_j) \textbf{ sy } B \not\simeq (E_i \| E_j) \textbf{ sy } A. \tag{5.4}$$

From the definition of type-p and $A \subseteq B$ it follows that $E_l \simeq E_l$ **sy** $B \simeq E_l$ **sy** A and $\mathsf{box}(E_l)$ **unsy** A is connected, for every $l \leq m$. Hence $(i,j) \in \rho$ if and only if $i \neq j$ and $\mathsf{box}((E_i \| E_j)$ **sy** $B)$ **unsy** A is a connected pre-box. The graph of the relation ρ can be divided into k connected components. Without loss of generality (one can always use STR4 and STR5) it may be assumed that $0 = m_0 < m_1 < \ldots < m_k = m$ are such that, for every $i \leq k$, the integers $1 + m_{i-1}, \ldots, m_i$ are the vertices of the i-th connected component of the graph of ρ. Denote $G_i = F_{1+m_{i-1}} \| \ldots \| F_{m_i}$, for every $i \leq k$. Clearly, for every $i \leq k$, $\mathsf{box}(G_i)$ **sy** B **unsy** A is connected. Thus, since $\mathsf{box}(E)$ **unsy** A is not connected, $k \geq 2$. It may then be observed that from proposition 55, corollary 8 and $A \subseteq B$, it follows that

$$((G_1 \textbf{ sy } B) \| \ldots \| (G_k \textbf{ sy } B)) \textbf{ sy } B \simeq ((G_1 \textbf{ sy } B) \| \ldots \| (G_k \textbf{ sy } B)) \textbf{ sy } A.$$

Denote $F_i = G_i$ **sy** B, for every $i \leq k$, and $F = (F_1 \| \ldots \| F_k)$ **sy** A. Therefore $E \equiv (F_1 \| \ldots \| F_k)$ **sy** $B \simeq (F_1 \| \ldots \| F_k)$ **sy** A. Hence, by proposition 70 and PROP3, $E \equiv F$.

(2,3) Proceed similarly as in the case of type-p expression. The main difference is that in the case of type-s expression, $0 = m_0 < m_1 < \ldots < m_k = m$ are assumed to be integers such that, for every $i \leq k$, there is a path in the graph of ρ between $1 + m_{i-1}$ and m_i and, for every $1 \leq j < k$, there is no path between x and y if $x \leq m_i < y$. □

Not every expression in Exp_0 can be rewritten into a normal form box expression. For example, if $E = ((a;\emptyset) \,\square\, \widehat{a})$ **sy** $\mathsf{a} \,\square\, (\emptyset; a)$ then $\mathsf{max}_E = \mathcal{A} - \mathsf{a}$ and the only decomposition of $\mathsf{box}(E)$ **unsy** $(\mathcal{A} - \mathsf{a}) = \mathsf{box}(E)$ into boxes Σ_i which could satisfy one of the parts of the definition of normal form expression, are the three nets shown in Figure 5.11 (note that $\mathsf{box}(E)$ **unsy** $(\mathcal{A} - \mathsf{a}) =$

$\Sigma_1 \,\square\, \Sigma_2 \,\square\, \Sigma_3$). While the first two nets do not create any problems, the third one does, as it is easy to see that there is no box expression E_3 such that Σ_3 is duplication equivalent to $\mathsf{box}(E_3)$ unsy A, for any synchronisation set A. Hence E has no normal form in the sense defined above. Therefore the applicability of the choice operator needs to be restricted. The definition below is motivated by the way in which the fourth case in the definition of normal form has been formulated (and, indirectly, by the characterisation of the situation when the unsy operator distributes over choice).

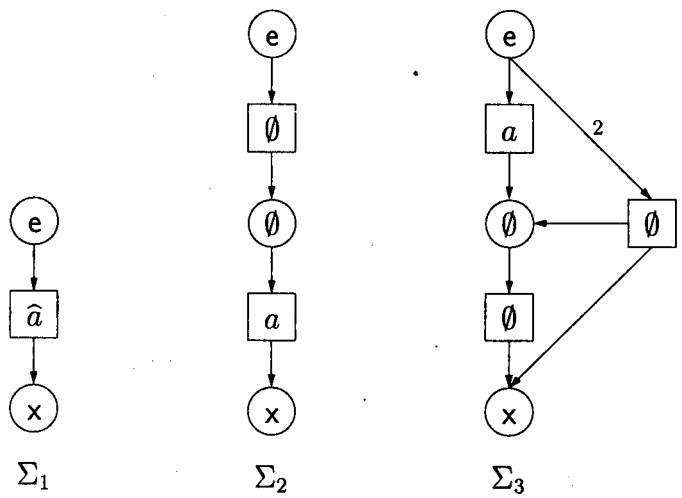


Figure 5.11: Decomposition of a de-synchronised box.

A box expression $E \in \mathsf{Exp}_0$ is *choice-restricted* if every subexpression F of E which has choice as the topmost operator is of the form $\alpha_1 \,\square\, \ldots \,\square\, \alpha_k \,\square\, H$ and satisfies ex_H **sy** $_A = \emptyset$, where A is the union of all the synchronisation sets B such that F lies within the scope of an application of **sy** B ($k = 0$ is allowed, and H may be missing if $k > 0$).[6] Then, let Exp_1 denote the set of all box expressions $G \in \mathsf{Exp}_0$ such that $G \equiv E$, for some choice-restricted box expression E. Note that the definition of Exp_1 is not fully syntactic. However, one can give simple syntactic conditions which guarantee that a box expression which is not choice-restricted belongs to Exp_1.

[6]Note that H can be an expression whose main connective is choice.

Proposition 78 If E is an expression in Exp_1 then there is an expression in normal form F such that $E \equiv F$.

Proof: It may be assumed that E is choice-restricted, and then proceed by induction on the number of transitions in the duplication quotient of $\langle E \rangle$. The base case is $\langle E \rangle = \alpha$ or $\langle E \rangle = \alpha \,\Box \ldots \Box\, \alpha$, for some action α. Then $E \equiv \alpha$ or $E \equiv \alpha \,\Box \ldots \Box\, \alpha$ which can be shown using PROP1, PROP2, PROP3 and PROP5. Moreover, in the latter case, using DUPL, STR2 and STR3, $\alpha \,\Box \ldots \Box\, \alpha$ may be transformed into α. In the inductive step a number of cases are considered, depending on the form of $\langle E \rangle$.

Case 1: $\langle E \rangle = E' \| E''$. By PROP3, PROP6, and proposition 70, $E \equiv (F \textbf{ sy } A \| G \textbf{ sy } A) \textbf{ sy } A$, where $A = \mathsf{max}_E$ and $F, G \in \mathsf{Exp}_1$. By the induction hypothesis, $E \equiv (F_0 \| G_0) \textbf{ sy } A$, where $F_0 \equiv F \textbf{ sy } A$ and $G_0 \equiv G \textbf{ sy } A$, and both F_0 and G_0 are in normal form. If both $\mathsf{box}(F_0) \,\mathsf{unsy}\, A$ and $\mathsf{box}(G_0) \,\mathsf{unsy}\, A$ are connected, then the result is shown since, by proposition 71(2,4,6),

$$\begin{aligned} \mathsf{box}(E) \,\mathsf{unsy}\, A &\simeq \mathsf{box}((F_0 \| G_0) \textbf{ sy } A) \,\mathsf{unsy}\, A \\ &= \mathsf{box}(F_0 \| G_0) \,\mathsf{unsy}\, A = (\mathsf{box}(F_0) \,\mathsf{unsy}\, A) \| (\mathsf{box}(G_0) \,\mathsf{unsy}\, A). \end{aligned}$$

Otherwise, without loss of generality, it may be assumed that $\mathsf{box}(F_0) \,\mathsf{unsy}\, A$ is not connected and $G_0 \,\mathsf{unsy}\, A$ is connected. Then, by proposition 77(1), $F_0 \equiv (F_1 \| \ldots \| F_k) \textbf{ sy } A$ where $k \geq 2$, $A \subseteq \mathsf{max}_{F_i}$ and $\mathsf{box}(F_i) \,\mathsf{unsy}\, A$ is connected, for every $i \leq k$. The induction hypothesis may be applied k-times to obtain that $E \equiv ((H_1 \| \ldots \| H_k) \textbf{ sy } A \| G_0) \textbf{ sy } A$, where each H_i is a normal form expression for every F_i. Next, by PROP3 and PROP5, $E \equiv (H_1 \| \ldots \| H_k \| G_0) \textbf{ sy } A$ which completes the discussion of this case since, by proposition 71(2,4,6),

$$\begin{aligned} \mathsf{box}(E) \,\mathsf{unsy}\, A &\simeq \mathsf{box}((H_1 \| \ldots \| H_k \| G_0) \textbf{ sy } A) \,\mathsf{unsy}\, A \\ &= \mathsf{box}(H_1 \| \ldots \| H_k \| G_0) \,\mathsf{unsy}\, A \\ &= (\mathsf{box}(H_1) \,\mathsf{unsy}\, A) \| \ldots \| (\mathsf{box}(H_k) \,\mathsf{unsy}\, A) \\ &\quad \| (\mathsf{box}(G_0) \,\mathsf{unsy}\, A). \end{aligned}$$

Case 2: $\langle E \rangle = [E' * E'' * E''']$. By PROP3, PROP6, and proposition 70, $E \equiv [F \textbf{ sy } A * G \textbf{ sy } A * H \textbf{ sy } A] \textbf{ sy } A$, where $A = \mathsf{max}_E$ and $F, G, H \in \mathsf{Exp}_1$. By the induction hypothesis, $E \equiv [F_0 * G_0 * H_0] \textbf{ sy } A$, where F_0, G_0 and H_0 are in normal form and $F_0 \equiv F \textbf{ sy } A$, $G_0 \equiv G \textbf{ sy } A$ and $H_0 \equiv H \textbf{ sy } A$.

Case 3: $\langle E \rangle = E'; E''$. Proceed similarly as in Case 1.

Case 4: $\langle E \rangle = E' \,\square\, E''$. By the definition of Exp_1, PROP3, PROP6, and proposition 70, $E \equiv (\alpha_1 \,\square \ldots \square\, \alpha_k \,\square\, G \textbf{ sy } A) \textbf{ sy } A$, where $G \in \mathsf{Exp}_1$, $A = \mathsf{max}_E$ and $\mathsf{ex}_{G \textbf{ sy } A} = \emptyset$. Moreover, by DUPL, it may be assumed that $\alpha_i \neq \alpha_j$, for every $i \neq j$. By the induction hypothesis, $E \equiv (\alpha_1 \,\square \ldots \square\, \alpha_k \,\square\, G_0) \textbf{ sy } A$, where $G_0 \equiv G \textbf{ sy } A$ and G_0 is in normal form. If $G_0 \,\mathsf{unsy}\, A$ is not choice-decomposable, then the result is shown, since, by proposition 73,

$$\begin{aligned}
\mathsf{box}(E) \,\mathsf{unsy}\, A &\simeq \mathsf{box}((\alpha_1 \,\square \ldots \square\, \alpha_k \,\square\, G_0) \textbf{ sy } A) \,\mathsf{unsy}\, A \\
&= \mathsf{box}((\alpha_1 \,\square \ldots \square\, \alpha_k \,\square\, G_0)) \,\mathsf{unsy}\, A \\
&= (\mathsf{box}(\alpha_1) \,\mathsf{unsy}\, A) \,\square \ldots \square\, (\mathsf{box}(\alpha_k) \,\mathsf{unsy}\, A) \\
&\quad \square\, (\mathsf{box}(G_0) \,\mathsf{unsy}\, A) \,\square\, (\mathsf{box}(G_0) \,\mathsf{unsy}\, A).
\end{aligned}$$

Otherwise, proposition 77(3) may be applied, and then proceed similarly as in case 1. □

Finally, it is possible to prove the main result of the second part of this section.

Theorem 7 For all box expression E, F in Exp_1, if $E \simeq F$ then $E \equiv F$. □

Proof: Follows from propositions 76 and 78. □

5.11 Conclusion

A sound and complete axiomatisation of duplication equivalence has been developed for a subset of box expressions. In doing so, it turned out that a crucial problem to be solved was that of a structural characterisation of maximal synchronisation sets of box expressions. It has been demonstrated

that such a characterisation is rather complicated for box expressions whose main connective (other than synchronisation) is the choice composition. This has led to a restriction on the set of box expressions for which the soundness and completeness results directly apply. The duplication equivalence is a very strong notion of equivalence which resembles the strong equivalence of CCS [41]. It is therefore natural to envisage that the future research will be concentrated on developing an axiomatisation of a weaker equivalence on box expressions, similar to the observational congruence of CCS. From this point of view the results obtained here are highly relevant since any axiomatisation of a weaker behavioural equivalence would encompass the axiomatisation of duplication equivalence. Moreover, the restrictions imposed on the type of box expressions for which the soundness and completeness results hold seem to be rather mild when considering a weaker notion of equivalence. Without going into details, if $F_1, \ldots, F_k$ are the subexpressions of a box expression E which cause the latter not to belong to Exp_1 then it should be possible (under any reasonable notion of observational equivalence which ignores internal moves) to replace each F_i by $F_i; \emptyset$, within E, and the resulting expression, call it $E^{\langle\emptyset\rangle}$, would now belong to Exp_1. It is also conjectured that the same transformation can be used to extend in a somewhat unusual way the completeness result obtained here, in the following way. If E and F are arbitrary box expressions such that $E \simeq F$ then $E^{\langle\emptyset\rangle} \equiv F^{\langle\emptyset\rangle}$, where it is assumed that for a box expression E in Exp_1, $E^{\langle\emptyset\rangle} = E$.

The final remark concerns the non-standard way in which some of the axioms were formulated since they refer to various sets (even sets of sets) of actions, such as covall. The reader might question whether this leads to a significant increase in the algorithmic complexity of the axiomatisation developed here when compared, e.g., with that presented in [41]. The answer is that it does not, as it is not difficult to see that all the sets involved are 'small' which

is due to an easy observation that it is always the case that

$$\sum_{A \in \mathsf{ccall}_E} |A| \leq k \quad \text{and} \quad |\mathsf{ex}_E| \leq k$$

where k is the number of action occurrences in a box expression E.

Chapter 6

Conclusion

In this chapter, a summary of the main results of the thesis are given, together with a discussion of possible areas for future investigation, building upon the work of the previous chapters.

6.1 Summary of Results

In Chapter 3 a detailed investigation into the synthesis and axiomatisation problems was carried out for a basic subset of the Petri Box Calculus shown in Table 6.1.

$E ::=$	α	Atomic action
	$\mid E \parallel E$	Parallel composition
	$\mid E \,\square\, E$	Choice composition
	$\mid E; E$	Sequential composition
	$\mid [E * E * E]$	Iteration

Table 6.1: Basic box expression syntax

Efficient algorithms for the problems listed in Table 6.2 were presented in Chapter 3. The time complexities of these algorithms is given in terms of n, the number of nodes in the input net, and a, the number of atomic actions

in the input expression. In addition, it has been shown that for any box expression, E, the number of expressions, E', such that $\mathsf{box}(E') = \mathsf{box}(E)$ can be calculated.

Problem	Time complexity
BOX EXPRESSION SYNTHESIS	$O(n^5)$
CANONICAL BOX EXPRESSION SYNTHESIS	$O(n^5)$
CANONICAL BOX EXPRESSION	$O(a^2 \cdot loga)$
PETRI BOX ISOMORPHISM	$O(n^5)$
BOX EXPRESSION ISOMORPHISM	$O(a^2 \cdot loga)$
BOX EXPRESSION ISOMORPHISM PROOF	$O(a^3)$

Table 6.2: Time complexity for basic syntax algorithms

Based on the framework provided by the synthesis algorithm, the axiom system in Table 3.6 was shown to be complete. In addition, a proof strategy for applying the axioms was presented, which allows the automatic generation of proofs.

In Chapter 4, the synthesis and axiomatisation problems were extended to the syntax in Table 6.3, which includes the synchronisation operator. The synthesis problem for this syntax was shown to be NP-hard. However, when the synthesised expression is allowed to contain the scoping operator in place of synchronisation, the extra expressiveness means that the synthesis problem is no longer NP-hard, and has an efficient solution.

The various algorithms investigated in Chapter 3 were revisited in Chapter 4. It was found that the extra work involved in the synthesis of synchronisation does not affect the overall time complexity of the algorithm, which remains at $O(n^5)$. An efficient algorithm was not found for the problem of synthesising a canonical form expression, and the related problems of rewriting an expression into canonical form and generating a proof of equivalence. Nor were these problems shown to be NP-hard. Instead, some evidence was

$E ::=$	α	Atomic action
	$E \parallel E$	Parallel composition
	$E \,\square\, E$	Choice composition
	$E; E$	Sequential composition
	$[E * E * E]$	Iteration
	E **sy** A	Synchronisation

Table 6.3: Box expression syntax with synchronisation

presented that the time complexity of finding a canonical form expression is related to the complexity of the graph isomorphism problem.

As in Chapter 3, the framework of the synthesis algorithm provided the basis for the production of an axiomatisation and aided the proof of completeness. The work in Chapter 3 was almost totally reused in the investigation into synchronisation.

In Chapter 5, consideration was given to extending the results for the syntaxes in Table 6.1 and Table 6.3 from the domain of net isomorphism to that of duplication equivalence. The existing results for isomorphism were reused, and only a minimal amount of work was required to extend the algorithms and axiomatisation to the domain of duplication equivalence.

Section 5.4 provides an investigation into the axiomatisation of the syntax in Table 6.3, where the framework for the synthesis algorithm is not reused. This work provides a contrasting approach to the axiomatisation problem. The investigation is motivated by the fact that the NP hardness result of Chapter 4 no longer holds when the net semantic is changed from isomorphism to duplication equivalence. Also, despite its location, the work in Section 5.4 was carried out in parallel and completed before the investigation into synchronisation in Chapter 4. The two approaches to finding an axiomatisation for the syntax in Table 6.3 for duplication equivalence demonstrate some of the benefits of re-use provided by the framework of the synthesis algorithm.

6.2 Extensions and Areas for Further Investigation

The framework provided by the synthesis algorithm allows the investigation into the synthesis and axiomatisation problems to be tackled in a modular fashion, based on particular subsets of the Petri Box Calculus, and particular notions of equivalence. The two main areas for further work involve extending the subset of the Box Algebra considered in this thesis, and investigating the problem for further notions of equivalence, particularly behavioural equivalences. In this section, some notes on these problems together with ideas for other avenues of further work are presented.

6.2.1 Additional Operators

In this section, some observations are made on the problem of extending the investigation into synthesis and axiomatisation to deal with the operations of restriction, scoping and recursion.

Restriction

Restriction can be regarded as a global operator, as it can affect any of the transitions in the net it operates on. It is different from the other operators in the Petri Box Calculus in that it is destructive in nature. The structure of a net may be radically changed by the application of the restriction operator. From this point of view, the top-down approach to the synthesis algorithm, which relies on the constructive nature of the semantics for box expressions, does not appear to be particularly suited to dealing with the restriction operator.

The simplest example of a restriction expression is $E = a$ **rs** a. The expression E can be rewritten in constructive terms as $E =$ **stop**. The second form for E is constructive because the a labelled transition that is removed in a **rs** a is never created in **stop**. This scheme can be extended to any ba-

sic syntax expression involving restriction, simply by replacing every atomic action that is restricted by **stop**, and removing all of the **rs** operators. The following axioms may be used to rewrite an expression involving the restriction operator, but not the synchronisation operator, into **stop** form.

$$
\begin{aligned}
E \textbf{ rs } A \textbf{ rs } B &= E \textbf{ rs } (A \cup B) \\
\alpha \textbf{ rs } A &= \begin{cases} \alpha & \text{if } \forall a \in A : \alpha \cap \{a, \widehat{a}\} = \emptyset \\ \textbf{stop} & \text{otherwise} \end{cases} \\
(E_1 \parallel E_2) \textbf{ rs } A &= E_1 \textbf{ rs } A \parallel E_2 \textbf{ rs } A \\
(E_1 \,\square\, E_2) \textbf{ rs } A &= E_1 \textbf{ rs } A \,\square\, E_2 \textbf{ rs } A \\
(E_1; E_2) \textbf{ rs } A &= E_1 \textbf{ rs } A; E_2 \textbf{ rs } A \\
[E_1 * E_2 * E_3] \textbf{ rs } A &= [E_1 \textbf{ rs } A * E_2 \textbf{ rs } A * E_3 \textbf{ rs } A]
\end{aligned}
$$

From the point of view of synthesis, there are still several problems, the greatest of which is the fact that the properties used in Chapter 3 to identify which synthesis rule to apply are no longer valid. For example, Figure 6.1, shows the implementation of a sequence expression, $E = a; \textbf{stop}; c$. The net in Figure 6.1 is disjoint, so would be identified by the standard synthesis algorithm as being the implementation of an expression whose main connective is parallel composition.

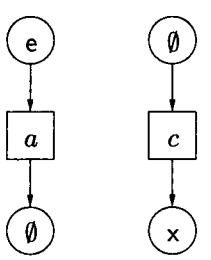


Figure 6.1: Disjoint net obtained from a sequence expression

Figure 6.2, shows a net which is the implementation of any of the following expressions:

$$E_1 = (a; \textbf{stop}; c) \parallel (b; \textbf{stop}; d)$$

$$E_2 \;=\; (a; \mathbf{stop}; d) \parallel (b; \mathbf{stop}; c)$$

The net in Figure 6.2 is also duplication equivalent to an implementation of:

$$E_3 = (a \parallel b); \mathbf{stop}; (c \parallel d)$$

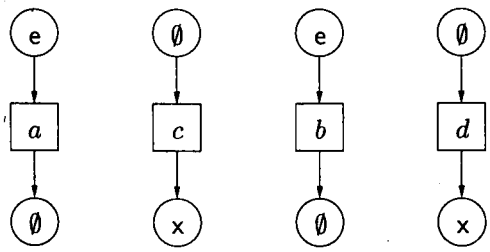


Figure 6.2: Problem of matching subnets

A partial investigation into the restriction operator leads to the following crucial observation. The implementation of the expression $E = \mathbf{stop}$ consists of a single isolated entry place, and a single isolated exit place. If the expressiveness of the Box Calculus is modified slightly so that it is possible to represent the isolated entry and exit places independently of each other, then it seems a large part of synthesis algorithm of Chapter 3 can be reused – in particular, every disjoint input net can be synthesised as an expression whose main connective is parallel composition.

The natural way to represent isolated places, especially in domain of isomorphism (where the numbers of each type of isolated place is significant), is to introduce an isolated places operator. The new operator has the syntax $\bigcirc_B^A$, and a semantics that creates A isolated entry places, B isolated internal places and C isolated exit places.

Using the isolated places operator, the net in Figure 6.1 could be represented by the expression

$$E = (a; \bigcirc_0^1) \parallel (\bigcirc_1^0 ; c)$$

and the net in Figure 6.2 by

$$E = (a; \bigcirc_{0}^{1}{}_{0}) \parallel (b; \bigcirc_{0}^{1}{}_{0}) \parallel (\bigcirc_{0}^{0}{}_{1}; c) \parallel (\bigcirc_{0}^{0}{}_{1}; d)$$

Of course the synthesis rules for choice, sequence and iteration would need to be modified to cope with restriction, although it seems that the modifications would take the form of extensions rather than replacement by completely new rules.

It would appear that moving from the domain of isomorphism to that of duplication equivalence may simplify the axiomatisation and synthesis problems due to the fact that the presence of restriction permits duplicated places to be generated, and the move to duplication equivalence removes the significance of the number of duplicates of each place.

Scoping

The semantics for the scoping operator are given syntactically in terms of the synchronisation and restriction operators:

$$[a : E] = E \textbf{ sy } a \textbf{ rs } a \tag{6.1}$$

This means that any synthesis algorithm that deals fully with restriction and synchronisation will automatically work for nets derived from expressions involving the scoping operator. Any complete axiomatisation that includes support for synchronisation and restriction can be extended to the scoping operator by adding the axiom (6.1).

Recursion

Expressions which involve the recursion operator generally produce infinite nets. For this reason, the framework relating the synthesis and axiomatisation problems is not suitable for dealing with recursion. An approach such as using fix-points would be required for axiomatising recursion. However, from a pragmatic view, the iteration operator provides the capability for infinite

behaviour, and in fact the translation from $B(PN)^2$ and OCCAM to Box expressions does not require the use of the recursion operator. In this respect, the Box Algebra can be considered expressive enough without the recursion operator.

6.2.2 Behavioural Equivalences

It seems unlikely that the framework in its present form can be used to solve the synthesis and axiomatisation problems, once the move is made from structural equivalences to behavioural equivalences. However, it may be possible to reuse the results for isomorphism and duplication equivalence in any investigation into behavioural equivalences.

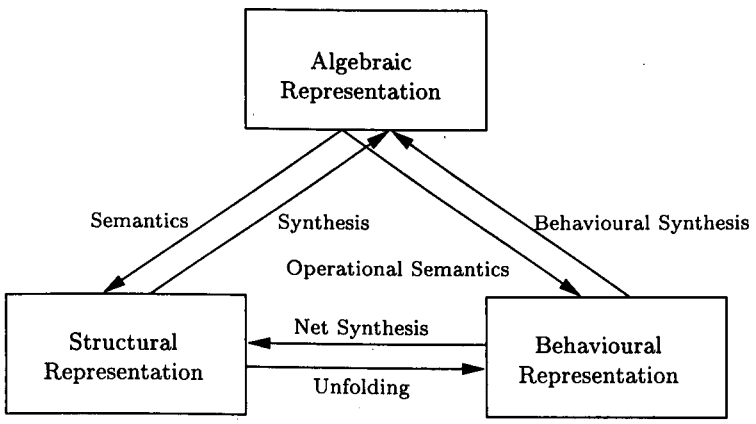


Figure 6.3: Synthesis for Behavioural Equivalences

Figure 6.3 illustrates a framework in which the synthesis and axiomatisation problems may be investigated for behavioural equivalences. As before, the axiomatisation would be derived as a result of a detailed analysis of the synthesis algorithm. There will be three different domains involved in any investigation:

- **Algebraic representation:** The domain of Box expressions.
- **Structural representation:** The domain of Petri Boxes.

- **Behavioural representation:** Some structural representation of the behaviour of a net, such as a net unfolding.

The work in this thesis concentrates on the algebraic and structural representations only, where a structural representation may be constructed from an algebraic one using the semantics of Box expressions, and the algebraic representation derived from a structural one using the synthesis algorithm.

A representation of the behaviour of a net may be constructed from the net itself, for example by unfolding the net. Through transitivity, this gives a representation for the behaviour of a Box expression. It may also be possible, using an operational semantics for the Box Algebra, to directly create the representation of the behaviour of an expression.

There are two possible approaches to a synthesis algorithm for a behavioural equivalence. The first is to synthesise a net from the representation of the behaviour of the system. The important point here, is that the synthesised net must be structurally equivalent to the implementation of a Box expression. Such an approach would require an investigation into the types of structures that arise in the behavioural representation as a result of particular constructs in the Box Calculus. The second method would be to synthesise a Box expression directly from the behavioural representation.

It is preferable that the behavioural representation is flexible enough to represent the behaviour of an arbitrary net. In this way the possibility of synthesising expressions for nets that are not implementations of Box expressions becomes feasible.

In the work for structural net equivalences, it was found that it was possible to reuse results when moving from one equivalence to another. It is expected that a similar re-use could be taken advantage of in the domain of behavioural equivalences. In [45], a two dimensional relationship between various net equivalences is illustrated, for example dividing equivalences into step semantics, interleaving semantics and partial order semantics. It is hoped that, for example, an investigation into partial order semantics would be applicable

to the various flavours of partial order semantics.

6.2.3 Net Based Operations

The framework proposed for the synthesis algorithm uses a set of structural properties of nets used to identify which of several synthesis rules to apply. These structural properties are, in part, based on properties of the net based operators, $\sqcup, \oplus, \ominus$ and $\otimes$, and their form of usage in describing the semantics of box expressions.

For example, usage of the $\sqcup$ operator, which directly corresponds to parallel composition, can be identified by checking whether the net is connected or not. Similarly, the definition of clusters of places matches the usage of the construction $\oplus(S_1 \otimes S_2)$ (where S_1 and S_2 are sets of places) in the semantics of box expressions. It is therefore, in some ways, not surprising that it was possible to reuse much of the work of Chapter 3 when extending the synthesis algorithm and axiomatisation to the syntax in Table 6.3.

One observation is that it shouldn't be too difficult to extend the synthesis algorithm and axiomatisations of Chapters 3 and 4 to cope with new operators defined in terms of the $\sqcup, \oplus, \ominus$ and $\otimes$ net based operators.

Given a particular notion of equivalence, it may be possible to axiomatise the properties of the net based operators. These axioms could be considered as meta-axioms, and be used to derive properties of the operators in the Box Calculus which are defined in terms of the net operators.

6.2.4 Alternative semantics

One of the problems with the current semantics for the Box Algebra is that the size of the implementation of a Box expression may be exponential in the size of the expression itself. A benefit of having a complete axiomatisation is that it provides a basis for checking designs of alternative semantics for the Box Algebra.

Given a new semantics, it would be necessary to show that all the properties encoded by the axiomatisation still hold, and that no additional properties hold (*i.e.* expressions that are not equivalent for the standard semantics do not become equivalent with the new semantics).

The main motivation for constructing a new semantics is to make the semantics more efficient in terms of the size of the representation. However, care is needed because it is unlikely that any new semantics will be consistent with the standard semantics over the whole range of possible equivalence relations. It is a different matter, of course, if a particular application requires only one notion of equivalence to be considered.

6.2.5 Time Complexity and Graph Isomorphism

An interesting point for further investigation from a complexity theoretic viewpoint is whether, for a particular notion of equivalence, the problem of checking the equivalence of a pair of box expressions has the same complexity (in terms of space and time) as checking the equivalence of an arbitrary pair of nets, (or, for structural equivalences, an arbitrary pair of graphs).

The investigation for isomorphism shows that, for example, checking equivalence of basic syntax box expressions is less complex than checking equivalence of an arbitrary pair of nets. However, when the synchronisation operator is added to the basic syntax, the problems of checking equivalence of a pair of expressions, and an arbitrary pair of graphs seems to become equally complex. When scoping is added to the syntax, the problem of checking equivalence of a pair of expressions provably becomes as difficult as the graph isomorphism problem.

The graph isomorphism problem is one of a small number of problem for which it is not known whether a tractable solution exists. Applying some of the ideas from the investigation into the synthesis problem for synchronisation and scoping may provide enough insight into the graph isomorphism problem to allow the question of its complexity to be resolved.

6.3 Conclusion

The work in this thesis has demonstrated a general approach to the the synthesis and axiomatisation problems for various subsets of the Petri Box Calculus for the structural equivalences isomorphism and duplication equivalence.

To an extent the investigation took a pragmatic approach which resulted in efficient algorithms for synthesis and in some cases the generation of proofs. In this respect, the algorithms presented may be suitable for inclusion in a modelling and verification tool such as PEP.

It is also believed that the work here could provide a solid basis for an investigation into the axiomatisation and synthesis problems for behavioural equivalences, such as a partial order semantics.

Appendix A

Definitions

This appendix provides cross references for the main definitions and concepts that have been introduced. The list of definitions are categorised into the following areas:

- Multisets
- Actions and basic actions
- Box expressions
- Classes of expressions
- Nets and net operators
- Equivalence of nets/expressions
- Ordering of nodes and expressions
- Sets of nodes
- Connectedness of nodes
- Equivalence of nodes
- Classes of nodes
- Action/transition mapping

- Synchronisation transitions
- Synchronisation sets
- Construction of maximal sy-sets

A.1 Multisets

multiset	Multisets: Section 1.3.1, Page 19.
$\cup$	Multiset union: Section 1.3.1, Page 20.
$\cap$	Multiset intersection: Section 1.3.1, Page 20.
$-$	Multiset difference: Section 1.3.1, Page 20.
$+$	Multiset sum: Section 1.3.1, Page 20.
$\cdot$	Multiset multiplication: Section 1.3.1, Page 20.
$\mid$	Multiset restriction: Section 1.3.1, Page 20.

A.2 Actions and Basic Actions

$\widehat{\ }$	Conjugation of basic actions: Section 1.2, Page 12.
$\mathcal{A}(\alpha)$	Unique word generated from an atomic action, α: Section 2.5.6, Page 84.
$<_A$	Ordering of atomic actions: Section 2.5.6, Page 84.
$<_b$	Ordering of basic actions: Section 2.5.6, Page 84.
μ	Labelling function which associates atomic actions with action names: Section 2.5.7, Page 85.
A^a	The set of action names whose label contains the basic action a or $\widehat{a}$: Section 4.6.4, Page 254.

ex_Σ The set of labels of all ex-transitions of Σ: Section 5.6.1, Page 305.

$\mathcal{L}(E)$ The set of basic actions appearing in the expression, E: Section 4.1, Page 167.

A.3 Box Expressions

α Atomic action: Section 1.2, Page 13 - informal description of intended behaviour, Section 1.3.5, Page 29 - formal semantics.

$E_1 \parallel E_2$ Parallel composition: Section 1.2, Page 13 - informal description of intended behaviour, Section 1.3.5, Page 29 - formal semantics.

$E_1 \,\square\, E_2$ Choice composition: Section 1.2, Page 13 - informal description of intended behaviour, Section 1.3.5, Page 29 - formal semantics.

$E_1; E_2$ Sequential composition: Section 1.2, Page 14 - informal description of intended behaviour, Section 1.3.5, Page 30 - formal semantics.

$[E_1 * E_2 * E_3]$ Iteration: Section 1.2, Page 14 - informal description of intended behaviour, Section 1.3.5, Page 30 - formal semantics.

E **rs** a Restriction: Section 1.2, Page 14 - informal description of intended behaviour, Section 1.3.5, Page 31 - formal semantics.

E **sy** a Synchronisation: Section 1.2, Page 15 - informal description of intended behaviour, Section 1.3.5, Page 32 - for-

mal semantics.

$[a : E]$	Scoping: Section 1.2, Page 16.
stop	Stop box: Section 1.2, Page 16.
$E[F]$	Relabelling operator: Section 1.2, Page 17.
$E_1[X \leftarrow E_2]$	Refinement: Section 1.2, Page 17 - informal description of intended semantics, Section 4.2.5, Page 186 - refinement is considered as a means of overcoming the NP hardness result for synchronisation synthesis.
$\mu X.E$	Recursion: Section 1.2, Page 18.
$\bigcirc_{B\ C}^{A}$	Isolated places operator: Section 6.2.1, Page 349.

A.4 Classes of Expressions

Exp_0	Syntactic restriction of the box expression syntax defined by (5.2) in Section 5.6: Section 5.6.1, Page 310.
Box_0	Restricted class of boxes corresponding to Exp_0: Section 5.6.1, Page 310.
Exp_1	Choice-restricted expression from Exp_0: Section 5.10.3, Page 339.

A.5 Nets and Net operators

$\mathsf{box}(E)$	Mapping from expressions to Petri Boxes: Section 1.3.5, Page 26.
Σ	Net $\Sigma = (S, T, W, \lambda)$: Section 1.3.2, Page 20.

implementation	A net, unique up to isomorphism, derived from a Box expression: Section 1.3.5.
$\sqcup$	Net union: Section 1.3.4, Page 24.
$\cup$	Net union operation for unionable nets: Section 5.4.1, Page 288.
$\ominus$	Operator to remove a set of nodes from a net: Section 1.3.4, Page 24.
$\oplus$	Operator to add a set of nodes to a net (Note: this operator has different definitions depending whether the set of nodes consists of places or transitions): Section 1.3.4, Page 25.
$\oplus$	Place replacement operator (alternative style of adding a set of places to a net): Section 5.5, Page 301.
Δ	Gluing set used in construction of boxes: Section 5.5, Page 302.
$\otimes$	Place multiplication: Section 1.3.4, Page 25.
$\uplus$	Net based operator to add a set of places to a net: Section 3.3, Page 98.
unsy	De-synchronisation operator: Section 5.10.2, Page 329.

A.6 Equivalence of Nets/Expressions

$[\Sigma]$	Equivalence class of nets: Section 1.3.5, Page 29.
$[E]_{iso}$	Class of isomorphic expressions: Section 1.4, Page 35.
$[E]_{dup}$	Class of duplication equivalent expressions: Section 1.4, Page 35.

$[E]_n$	Class of expressions equivalent with respect to the relation, n: Section 1.4, Page 35.
$=_{iso}$	Isomorphic: Section 1.4, Page 35.
$=_{dup}$	Duplication equivalent: Section 1.4, Page 35.
$[\Sigma]_{iso}$	Class of isomorphic nets: Section 1.4.1, Page 36.
$[\Sigma]_{dup}$	Class of duplication equivalent nets: Section 1.4.2, Page 37.
$[\Sigma]_{\simeq}$	Duplication quotient of net Σ: Section 5.4.1, Page 289.
$\Sigma_1 \cong \Sigma_2$	Place-preserving duplication equivalence of nets: Section 5.4.1, Page 291.

A.7 Ordering of Nodes and Expressions

$<_t$	Arbitrary fixed ordering of transitions: Section 2.5.6, Page 84.
$<_l$	Ordering of transitions in a net: Section 2.5.6, Page 84.
$\min(T)$	The smallest transition in the set T, with respect to the ordering, $<_l$: Section 2.5.6, Page 84.
$<_e$	Ordering over basic syntax expressions: Section 3.5.3, Page 147.
$Ord(E)$	Ordered standard form for basic syntax expressions: Section 3.5.3, Page 148.

A.8 Sets of Nodes

$^\bullet\Sigma$	Entry places of a net: Section 1.3.2, Page 21.

$\Sigma^\bullet$	Exit places of a net: Section 1.3.2, Page 21.
${}^\bullet n$	Pre-places of a node (place or transition): Section 2.5.1, Page 77.
$n^\bullet$	Post-places of a node (place or transition): Section 2.5.1, Page 77.
${}^\bullet N$	Pre-places of a set of nodes (places or transitions): Section 2.5.1, Page 77.
$N^\bullet$	Post-places of a set of nodes (places or transitions): Section 2.5.1, Page 77.
S_{e}	Entry places of a net: Section 2.5.1, Page 76.
S_{i}	Internal places of a net: Section 2.5.1, Page 76.
S_{x}	Exit places of a net: Section 2.5.1, Page 76.
N_{a}	The set of all nodes of a net: Section 2.5.1, Page 76.
N_{i}	The set of all internal nodes of a net: Section 2.5.1, Page 76.
T_{e}	The set of transitions directly connected to an entry place: Section 2.5.1, Page 76.
T_{x}	The set of transitions directly connected to an exit place: Section 2.5.1, Page 76.
$\mathcal{I}(\Sigma)$	The set of isolated places in a net: Section 2.5.1, Page 77.
EX-transition	Transition which has constant connectivity with every entry and exit place: Section 5.4.2, Page 299.
T_{EX}	The set of EX-transitions: Section 5.4.2, Page 299.
EX_Σ	The set of *ex*-transitions of Σ: Section 5.6.1, Page 305.

A.9 Connectedness of Nodes

$\overset{\leftrightarrow}{\sim}_N$ Undirected connectedness relation: Section 2.5.2, Page 77.

$\mathcal{G}(N)$ The set of connected components containing the set of nodes, N: Section 2.5.2, Page 78.

$\overset{\rightarrow}{\sim}$ Directed connectedness relation: Section 2.5.2, Page 79.

$t \bowtie T$ Connectivity relation: Section 2.5.4 , Page 81 - transition t has the same connectivity as the set of transitions, T, Section 5.4.1, Page 286 - extension of the definition to allow the connectivity of two sets of transitions to be compared.

δ A dummy transition which, if it were present, would connect to every entry and exit place of the net: Section 5.4.1, Page 286.

const_Σ Constant connectivity: Section 5.4.1, Page 287.

ex-path Connected sequence of nodes starting with an entry place and finishing with an exit place: Section 5.10.2, Page 331.

ex-connected A net is *ex*-connected if every place in the net belongs to an *ex*-path: Section 5.10.2, Page 331.

A.10 Equivalence of Nodes

$\sim_{dpl}$ Duplication equivalence of transitions (based on connectivity only, not labels): Section 2.5.4, Page 81.

$Dpl(t)$ The set of transitions which duplicate t (Note: these transitions do not necessarily have the same label as t): Section 2.5.4, Page 82.

$\simeq$ Duplication equivalence relation: Section 5.4.1, Page 287.

$\simeq_{\Sigma_1\Sigma_2}$ Duplication equivalence relation for nodes in place sharing nets Σ_1 and Σ_2: Section 5.4.1, Page 291.

$[n]_{\simeq}$ Equivalence class of duplication equivalent nodes which contains n: Section 5.4.1, Page 287.

A.11 Classes of Nodes

$\simeq_p$ Equivalence relation which partitions places into clusters: Section 2.5.3, Page 79.

$\mathcal{C}(s)$ The cluster of places to which s belongs: Section 2.5.3, Page 80.

$\mathcal{C}_{\mathsf{i}}(\Sigma)$ The set of clusters of internal places of the net, Σ: Section 2.5.3, Page 81.

$\sim_{\mathsf{e}}$ Equivalence classes of entry places arising from choice decomposition: Section 3.3.3, Page 103.

$P_{T_{\mathsf{e}}}$ Partitioning of entry places arising from choice decomposition: Section 3.3.3, Page 103.

$P_{T_{\mathsf{x}}}$ Partitioning of exit places arising from choice decomposition: Section 3.3.3, Page 103.

$S_;$ Clusters of places forming interfaces between subnets in sequence decomposition: Section 3.3.4, Page 107.

$<_s$ Ordering of the clusters of places in $S_;$: Section 3.3.4, Page 107.

S_{if} Clusters of places which form the interfaces between subnets in iteration decomposition: Section 3.3.5, Page 112.

$\otimes$-sets — Cluster of places created by the operation of place multiplication: Section 5.4.1, Page 288.

A.12 Action/Transition Mapping

ϕ — Mapping from an action name in an expression to the set of transitions derived from that action in the implementation of the expression: Section 2.5.7, Page 86.

$\sim_\phi$ — Equivalence class of transitions arising from the same action: Section 2.5.7, Page 89.

$\odot$ — Relates the synchronisation of actions in an expression with the synchronisation of transitions in the corresponding net: Section 2.5.8, Page 90.

A.13 Synchronisation Transitions

T^a — Set of transitions with an a or $\hat{a}$ in their label: Section 1.3.5, Page 27.

T_{sy} — Set of transitions created by a synchronisation operation on a net: Section 1.3.5, Page 32, Section 4.5.1, Page 217.

$T_{sc}(\Sigma)$ — The set of transitions that can be represented by the scoping operator in the synthesised expression: Section 2.5.5, Page 82.

T_{at} — The subset of T_{sc} that may be represented by atomic actions in the synthesised expression: Section 4.5.1, Page 217.

$T_b(t)$ — The base transitions of a transition, t: Section 2.5.5, Page 83.

syn_A The set of all pairs of A-synchronisable transitions: Section 5.4.2, Page 292.

$t_1 \diamond t_2$ Synchronisation of transitions t_1 and t_2: Section 5.4.2, Page 293.

A.14 Synchronisation sets

max_Σ Maximal synchronisation set: Section 5.4.2, Page 296.

simex_{EF} Set of synchronisation sets: Section 5.7, Page 315.

$\aleph$ A context: Section 5.10, Page 323.

$\langle E \rangle$ The expression resulting from the deletion of all synchronisation operations in E: Section 5.10, Page 324.

$\langle \aleph \rangle$ The context resulting from the deletion of all synchronisation operations in $\aleph$: Section 5.10, Page 324.

$A^i_\aleph$ The synchronisation set directly applied to the i-th place holder of the context $\aleph$: Section 5.10, Page 324.

$A^{ij}_\aleph$ The union of synchronisation sets, A, such that the i-th and j-th place holders of the context, $\aleph$ are in the scope of an application of **sy** A: Section 5.10, Page 324.

A.15 Construction of maximal sy-sets

ccall_Σ The set of labels of all choice context transitions: Section 5.6.1, Page 305.

ccint_Σ The set of labels of internal choice context transitions: Section 5.6.1, Page 305.

$\mathsf{covall}^{A}_{\Sigma_1\Sigma_2}$	Auxiliary notation used in the definition of maximal synchronisation sets: Section 5.6.1, Page 308.
ccnoex_{Σ}	The set of labels of internal choice context transitions satisfying some additional conditions: Section 5.6.1, Page 310.
$\mathsf{covnoex}^{A}_{\Sigma_1\Sigma_2}$	Auxiliary notation used in the definition of maximal synchronisation sets for choice composition: Section 5.6.1, Page 310.
$\mathsf{covmix}^{A}_{\Sigma_1\Sigma_2}$	Auxiliary notation used in the definition of maximal synchronisation sets for choice composition: Section 5.6.1, Page 310.
$\mathsf{covint}^{A}_{\Sigma_1\Sigma_2}$	Auxiliary notation used in the definition of maximal synchronisation sets for choice composition: Section 5.6.1, Page 310.

Appendix B

Subsets of the Petri Box Calculus

During the course of the investigations into the synthesis and axiomatisation problems for isomorphism and duplication equivalence, various subsets of the Petri Box Calculus have been used. In this appendix, a short description of the subset of the calculus, and a discussion of any restrictions is given for the following areas which were investigated in the previous chapters of this thesis:

- **Basic syntax, isomorphism**, Chapter 3.
- **Synchronisation synthesis, isomorphism**, Section 4.3.
- **Synchronisation axiomatisation, isomorphism**, Section 4.6.4.
- **Basic syntax, duplication equivalence**, Section 5.2.
- **Synchronisation, duplication equivalence (1st approach)**, Section 5.3.
- **Synchronisation, duplication equivalence (2nd approach)**, Section 5.4.

B.1 Basic syntax (isomorphism)

Chapter 3 presents an investigation into the synthesis and axiomatisation problems for isomorphism, restricted to the domain of the basic box expression syntax given in Table B.1.

$E ::=$	α		Atomic action
	$\mid$	$E \parallel E$	Parallel composition
	$\mid$	$E \,\square\, E$	Choice composition
	$\mid$	$E; E$	Sequential composition
	$\mid$	$[E * E * E]$	Iteration

Table B.1: Basic box expression syntax

Note: No restriction is placed on the form of atomic actions (*i.e.* multi-actions are allowed).

The input to the synthesis algorithm of Chapter 3 can be the implementation of any expression over the syntax in Table B.1. Similarly, the output of the synthesis algorithm is guaranteed to be a member of the box expression syntax of Table B.1.

The axiom system of Section 3.5.5 is closed with respect to the basic box expression syntax of Table B.1 – that is, applying any axiom from Table 3.6 to an expression from the syntax in Table B.1 will always result in a basic syntax box expression.

B.2 Synchronisation Synthesis (isomorphism)

The synthesis algorithm of Section 4.3 considers as input a net which is the implementation of an expression from a restricted class of the syntax shown in Table B.2.

Note: The semantics for the synchronisation operator used when constructing an implementation of an expression are slightly different from the

$E ::=$	α	Atomic action
	$E \parallel E$	Parallel composition
	$E \,\square\, E$	Choice composition
	$E; E$	Sequential composition
	$[E * E * E]$	Iteration
	E **sy** A	Synchronisation

Table B.2: Synchronisation synthesis box expression syntax

original semantics presented in [5]. The modified semantics ensure that duplicates of atomic actions which take part in a synchronisation are not created (these are significant for isomorphism, but not for the duplication equivalence used in [5]). See Section 1.3.5 for details.

B.2.1 Restriction of expression syntax

No restriction is placed on the form of atomic actions, and, in particular, multiactions are permitted. The class of input nets to the synthesis algorithm is restricted to finite nets. Hence, attention is restricted to those expressions over the syntax in Table B.2 whose implementation is finite. For example, this means that the expression $\{a, \widehat{a}\}$ **sy** a is not considered. Section 4.2.2 describes a procedure for detecting when an expression has an infinite implementation.

The reasoning behind restricting the class of expressions considered in this way is twofold:

- One of the aims of the investigation into the synthesis algorithm is the production of an efficient synthesis process. From an algorithmic viewpoint, it is impractical to consider infinite input nets.

- When an infinite synchronisation occurs, only a finite number of the infinite number of synchronised transitions can ever be enabled during execution of the net. While these transitions are significant for structural

semantics such as isomorphism, once behavioural semantics are considered, transitions that cannot be enabled are not significant, and can be ignored.

B.2.2 Form of synthesised expressions

The synthesis algorithm of Chapter 4 differs from the basic syntax synthesis algorithm in that the box expression syntax used for the synthesised expression is different from the syntax used to define the class of input nets. Table B.3 shows that for synthesised expressions, the scoping operator is used in place of synchronisation.

$E ::=$	α	Atomic action
	$\mid E \parallel E$	Parallel composition
	$\mid E \,\square\, E$	Choice composition
	$\mid E; E$	Sequential composition
	$\mid [E * E * E]$	Iteration
	$\mid [A : E]$	Scoping

Table B.3: Output box expression syntax

Clearly, the syntax of Table B.3 is more expressive than that of Table B.2 because the scoping operator has the expressive power of both the synchronisation and the restriction operators. However, the form of synthesised expressions produced by the synthesis algorithm of Chapter 4 is restricted as follows:

- Scoping operators may only appear immediately inside an iteration operator, or acting on the entire expression.
- No basic action is scoped more than once. That is, for any pair of scoping operations in the expression, scoping by sets of basic actions, N_1 and N_2, then $N_1 \cap N_2 = \emptyset$.

- For each basic action, n, that is scoped, there will be exactly one action labelled $\widehat{n}$, and no other action contains $\widehat{n}$ in its label. Furthermore, there will be exactly one action whose label contains the basic action, n.

B.3 Synchronisation Axiomatisation (isomorphism)

The aim of the axiom system presented in Section 4.6.4 is to provide the ability to rewrite a suitable expression, E, from the syntax in Table B.2 into the form of the synthesised expression that would be produced if the implementation of E were given as input to the synthesis algorithm of Chapter 4. As such, it is necessary for subset of the box calculus used by the axiom system to include both the synchronisation and scoping operators. This box expression syntax is given in Table B.4.

$E ::=$	α	Atomic action
$\mid$	$E \parallel E$	Parallel composition
$\mid$	$E \,\square\, E$	Choice composition
$\mid$	$E; E$	Sequential composition
$\mid$	$[E * E * E]$	Iteration
$\mid$	E **sy** A	Synchronisation
$\mid$	$[A : E]$	Scoping

Table B.4: Axiom system expression syntax

The restrictions of Section B.2.1 also apply to the expression syntax for the axiom system of Section 4.6.4. In addition, two of the three restrictions on the form of synthesised expressions apply to the axiom system expression syntax – namely:

- No basic action is scoped more than once. That is, for any pair of scoping operations in the expression, scoping by sets of basic actions, N_1 and N_2,

then $N_1 \cap N_2 = \emptyset$.

- For each basic action, n, that is scoped, there will be exactly one action labelled $\widehat{n}$, and no other action contains $\widehat{n}$ in its label. Furthermore, there will be exactly one action whose label contains the basic action, n.

Note that no restriction is placed on the position of the scoping operators, and there are axioms which allow those movements of scoping operators which preserve soundness.

B.4 Basic syntax (duplication equivalence)

Section 5.2 presents an investigation into the synthesis and axiomatisation problems for duplication equivalence, restricted to the domain of the basic box expression syntax given in Table B.1.

All the notes in Section B.1 equally apply to the synthesis algorithm and axiom system of Section 5.2.

B.5 Synchronisation (duplication equivalence) - Approach I

Section 5.3 gives a discussion on the synthesis and axiomatisation problems for duplication equivalence restricted to the domain of those box expressions from the syntax in Table B.2 whose implementation is a finite net.

The box expression syntaxes of Tables B.2, B.3, and B.3 are used respectively for defining the class of nets suitable as input to the synthesis algorithm, for the synthesised expression, and for the axiom system. Those restrictions to the various syntaxes described in Section B.2.1 also apply to the investigation in Section 5.3.

B.6 Synchronisation (duplication equivalence) - Approach II

The second approach to the investigation into an axiomatisation for duplication equivalence for the syntax in Table B.2 is different in that the scoping operator is not used. The axiom system is entirely within the domain of the box expression syntax of Table B.2.

The biggest restriction placed of the form of expressions (nets) is that atomic actions (transition labels) may consist of only a single basic action, or the empty action. Note that this restriction means that it is not possible to generate an infinite synchronisation, and so all nets derived from the restricted syntax are guaranteed to be finite.

A syntactic restriction on choice (sub)expressions is introduced, so that in $E_1 \,[]\, E_2$, the implementations, Σ_1, Σ_2 of E_1 and E_2 cannot contain transitions, t_1, t_2, labelled a and $\widehat{a}$ such that:

$$
\begin{aligned}
{}^\bullet t_1 &= {}^\bullet \Sigma_1 \\
t_1{}^\bullet &= \Sigma_1{}^\bullet \\
{}^\bullet t_2 &= {}^\bullet \Sigma_2 \\
t_2{}^\bullet &= \Sigma_2{}^\bullet
\end{aligned}
$$

This class of restricted expressions is named Exp_0, and the corresponding class of boxes, Box_0. The restriction is introduced to simplify the characterisation of maximal synchronisation sets for choice composition.

The applicability of the choice operator is restricted further to allow the de-synchronisation operator, unsy to distribute over choice. This restriction is described fully by the definition of the class of expressions, Exp_1 in Section 5.10.3, Page 339.

Bibliography

[1] J.C.M.Baeten, J.A.Bergstra *Non Interleaving Process Algebra* In Proceedings CONCUR'93, Springer-Verlag Lecture Notes in Computer Science Volume 715, 308-323 (1993).

[2] J.C.M.Baeten, W.P.Weijland: *Process Algebra.* Cambridge Tracts in Theoretical Computer Science, Volume 18 (1990).

[3] E.Best: *Partial Order Verification with PEP* Proceedings of Partial Order Methods in Verification (1996).

[4] E.Best, R.Devillers, J.Esparza: *General Refinement and Recursion Operators for the Petri Box Calculus.* Springer-Verlag Lecture Notes in Computer Science Volume 665, 130-140 (1993).

[5] E.Best, R.Devillers, J.Hall: *The Petri Box Calculus: a New Causal Algebra with Multi-label Communication.* Advances in Petri Nets 1992, Springer-Verlag Lecture Notes in Computer Science Volume 609, 21-69 (1992).

[6] E.Best, J.Hall: *The Box Calculus: a New Causal Algebra with Multi-label Communication.* Technical Report No. 373, Computing Laboratory, University of Newcastle upon Tyne (1992).

[7] E.Best, R.P.Hopkins: *B(PN)^2 – a Basic Petri Net Programming Notation.* Proceedings of PARLE-93, Springer-Verlag Lecture Notes in Computer Science Volume 694, 379-390 (1993).

[8] E.Best, H.Fleischhack, W.Frączak, R.P.Hopkins, H.Klaudel, E.Pelz: *A Class of Composable High Level Petri Nets.* Application and Theory of Petri Nets 1995, Springer-Verlag Lecture Notes in Computer Science Volume 935, 102-120 (1995).

[9] E.Best, H.Fleischhack, W.Frączak, R.P.Hopkins, H.Klaudel, E.Pelz: *An M-net Semantics of $B(PN)^2$* Proceedings of STRICT'95, Berlin, J.Desel (ed), Springer-Verlag, Workshops in Computing, 85-100 (1995).

[10] E.Best, B.Grahlmann: *PEP - more than a Petri Net Tool.* Tools and Algorithms for the Construction and Analysis of Systems, 2nd International Workshop, TACAS'96, Springer-Verlag Lecture Notes in Computer Science Volume 1055, 387-401 (1996).

[11] J.A.Bergstra, J.W.Klop: *Algebra of communicating processes with abstraction* Theoretical Computer Science 37, 77-121 (1985).

[12] E.Best, M.Koutny: *Solving Recursive Net Equations.* Proceedings of ICALP-95, Springer-Verlag Lecture Notes in Computer Science Volume 944, 605-623 (1995).

[13] E.Best, H.G.Linde-Göers: *Compositional Process Semantics of Petri Boxes* Proceedings of Mathematical Foundations of Programming Semantics, Springer-Verlag Lecture Notes in Computer Science Volume 802, 250-270 (1993).

[14] G.Boudol, G.Roucairol, R.De Simone: *Petri Nets and Algebraic Calculi of Processes* Advances in Petri Nets 1985, Springer-Verlag Lecture Notes in Computer Science Volume 222, 41-58 (1985).

[15] I.Borosch, L.B.Treybig: *Bounds on positive integral solutions of linear Diophantine equations* Proceedings of American Math. Society Volume 55, 299-304 (1976).

[16] T.Basten, M.Voorhoeve *An Algebraic Semantics for Hierarchical P/T Nets* Application and Theory of Petri Nets 1995 , Springer-Verlag Lecture Notes in Computer Science Volume 935, 45-65 (1995).

[17] S.Christensen: *Decidability and Decomposition in Process Algebras* Report ECS-LFCS-93-278, University of Edinburgh, Department of Computer Science (1993).

[18] C.Dietz, G.Schreiber: *A Term Representation of P/T Systems* Application and Theory of Petri Nets 1994 , Springer-Verlag Lecture Notes in Computer Science Volume 815, 239-257 (1994).

[19] J.Engelfriet: *Branching Processes of Petri Nets* Acta Informatica 28 (1991).

[20] J.Esparza: *Model Checking Using Net Unfoldings* Proceedings of TAPSOFT'93, Springer-Verlag Lecture Notes in Computer Science Volume 668, 613-628 (1993).

[21] P.Degano, R.De Nicola, U.Montanari: *A Distributed Operational Semantics for CCS Based on Condition/Event Systems.* Acta Informatica 26 (1-2), 59-91 (1988).

[22] H.Fleischhack, B.Grahlmann: *A Petri Net Semantics for $B(PN)^2$ with Procedures which Allows Verification* Hildesheimer Informatikbericht 21/96 (1996)

[23] R.J.Van Glabbeek: *A Complete Axiomatixation for Branching Bisimulation Congruence of Finite-State Behaviours*

[24] U.Goltz: *On Representing CCS Programs by Finite Petri Nets.* Arbeitspapiere der GMD 290 (1988).

[25] M.R.Garey, D.S.Johnson *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H.Freeman and Company (1979).

[26] R.J.Van Glabbeek, F.W.Vaandrager: *Petri Net Models for Algebraic Theories of Concurrency* PARLE'87 Volume II, Springer-Verlag Lecture Notes in Computer Science Volume 259, 224-242 (1987). Information Processing '89, 613-618 (1989).

[27] R.Van Glabbeek, W.Weijland: *Branching Time and Abstraction in Bisimulation Semantics.* Technical Report TUM-I9052, Institut für Informatik, Technische Universität München (1990).

[28] C.A.R.Hoare: *Communicating Sequential Processes* Prentice Hall (1985).

[29] J.G.Hall: *An Algebra of High-level Petri Nets* PhD Thesis, University of Newcastle upon Tyne (1996).

[30] J.Hall, R.P.Hopkins, O.Botti: *A Basic-Net Algebra for Program Semantics and its Application to OCCAM* Advances in Petri Nets 1992, Springer-Verlag Lecture Notes in Computer Science Volume 609, 179-214 (1992).

[31] INMOS: *OCCAM 2 Reference Manual* Prentice Hall (1988).

[32] M.Jantzen: *Language Theory of Petri Nets* in BRR'87 397-412 (1987).

[33] R.M.Karp: *Reducibility among combinatorial problems* Complexity of Computer Communications, Plenum Press, 85-103 (1972).

[34] L.G.Khachian: *A polynomial algorithm in linear programming* (English translation) Soviet Math. Dokl. Volume 20, 191-194 (1979).

[35] M.Koutny: *Partial Order Semantics of Box Expressions.* Proceedings of Application and Theory of Petri Nets 1992, Springer-Verlag Lecture Notes in Computer Science Volume 815, 318-337 (1994).

[36] M.Koutny, J.Esparza, E.Best: *Operational Semantics for the Petri Box Calculus* Proceedings of CONCUR'94 Springer-Verlag Lecture Notes in Computer Science Volume 836, 210-225 (1994).

[37] M.Koutny, E.Best: *Operational and Denotational Semantics for the Box Algebra.* Technical Report, Computing Laboratory, University of Newcastle upon Tyne (1995).

[38] H.Klaudel, E.Pelz: *Handling Abstract Data Types in the Petri Box Calculus* CS&P'94 (1994)

[39] J.Lilius, E.Pelz: *An M-net Semantics for $B(PN)^2$ with Procedures.* 11th International Symposium on Computer and Information Science, Antalya (1996).

[40] B.D.McKay: *Practical Graph Isomorphism* Congress Numerantium 30, 45-87 (1981).

[41] R.Milner: *Communication and Concurrency.* Prentice Hall (1989).

[42] R.Milner: *A complete axiomatisation for observational congruence of finite-state behaviours.* Information and Computation Volume 81, 227-247 (1989).

[43] U.Montanari, D.Yankelevich: *Combining CCS and Petri Nets via Structural Axioms* Fundamenta Informaticae 20 (1-3), 193-229 (1994).

[44] E.Olderog: *Petri Nets and Algebraic Calculi of Processes* Advances in Petri Nets 1987, Springer Verlag Lecture Notes in Computer Science Volume 266, 196-223 (1987).

[45] L.Pomello, G.Rozenberg, C.Simone: *A Survey of Equivalence Notions for Net Based Systems.* Advances in Petri Nets 1992, Springer-Verlag Lecture Notes in Computer Science Volume 609, 410-467 (1992).

[46] W.Reisig: *Petri Nets, An Introduction.* EATCS Monographs on Theoretical Computer Science, Volume 4, Springer-Verlag (1985).

[47] T.J.Schaefer: *The complexity of satisfiability problems.* Proceedings 10th Annual ACM Symposium on Theory of Computing, 216-226 (1978).

[48] D.Taubner: *Finite Representation of CCS and TCSP Programs by Automata and Petri Nets.* Springer-Verlag Lecture Notes in Computer Science, Volume 369 (1989).