Project FORFUN

transforming SCCS expressions to structures of finite-state machines

Brondijk, R.A.

Award date:
1987

PROJECT FORFUN

Transforming SCCS Expressions To Structures of Finite-State Machines

R. Brondijk

10 July 1987

Summary

This paper concerns a design language which can facilitate a designer in the development traject of a VLSI circuit. The goal is to offer the designer a formal language with which he can express his design as an abstract idea and as a sequential circuit. The formal language will enable the designer to reason about his design and to check it for correctnes. The formal language handled here is Milner's Sequential Calculus of Communicating Systems; SCCS.

In the VLSI design traject, the designer works towards a target structure; a sequential circuit. The designer, however, will usually not have a single finite-state machine in mind, but a circuit with structure. For this reason, the designer should be facilitated with a finite-state machine model with structure. In this paper, such a model is proposed, the model of Communicating Finite-State Machines(CFSM). This model allows a designer to compare a specification of a structure of finite-state machines with that of a single finite-state machine.

Given a target structure, it is crutial for a designer to know what a valid specification of a structure of finite-state machines is. To these ends, a transformation process is developed from a SCCS expression to a structure of CFSMs. The transformation process resulted in the definition of a subset of SCCS which maps to an implementable structure of finite-state machines. In the design process, a designer can now strive towards a SCCS specification of an implementable circuit.

Acknowledgements

This thesis is the result of a research project conducted in the Philips Research Laboratories in Eindhoven. This project constitutes the final phase of my studies at the Technical University Eindhoven in the department of Electrical Engineering.

I would like to thank everyone in the group of Drs. E.P.C. van Utteren for their support and inspiration. In particular, I would like to thank my supervisor at the Lab, Cees Niessen, for his invaluable advice and constructive criticism, which kept me on the right path and Ton Kalker, for his mathematical advice and guidance. Finally, my special thanks to my superviser at the University, Prof. Koomen, for his ideas and support and for making my research at this Lab possible.

Contents

Introduction

In this thesis, a transformation process will be developed from an SCCS expression to a specification of a structure of finite-state machines.

SCCS, Synchonous Calculus of Communicating Systems, is a formal language, developed by R.Milne, and will be introduced in Chapter 1. The specification of an individual finite-state machine in a structure will closely resemble that of a 'standard' FSM, which will be described in Chapter 2.

Before the transformation process is development, two very important questions will be addressed. The first concerns the expressive power of SCCS and the second concerns the underlying model of finite-state machines.

The expressive power of SCCS is reflected by the class of problems it can solve and by the 'elegance' of the language. In the first Chapter, it will be shown that its expressive power is equivalent to that of a Turing Machine. This implies that all computable problems can be encoded and solved in SCCS.

Elegance is often measured by comparing a 'new' language with well known languages and by the relative 'ease', with which certain 'difficult' problems can be described and solved. In Chapter 1, the 'elegance' will be studied by comparing the semantics of a subset of SCCS, CCS, to two similar languages, CSP and Occam. The problems tackled will be the description of a stack, a counter and various logic gates.

The second question to be addressed is what kind of FSM is used in the underlying model. The underlying model must contain the notion of structure. This notion goes hand in hand with that of 'connection', allowing machines to communicate. To stress the idea of structure, the underlying model will be called 'Communicating Finite-State Machines' (CFSM). This model will be introduced in Chapter 2.

The development of the transformation process will entail the develop-

ment of a relationship between SCCS and the underlying CFSM model. The SCCS description and its transform must possess the same meaning. In Chapter 3, a meaning preserving transformation will be developed. Its development is possible because both models share the same operational semantics, namely Transition Graphs. The definition of Transition Graphs and their equivalence relation, will be given in Chapter 1. The correctness of the tranformation will be proved by showing that the SCCS expression and its CFSM transform both have the same Transition Graph representation.

The given transformation process lays the foundation of an easy to implement description method and verification process of a logic circuit. SCCS contains a simple communication model. Given the SCCS description of a structure of CFSM, it can be used to formally generate the specification of the entire structure from the specification of the individual components. Verification is then only a question of proving the generated specification equivalent to the given specification of the circuit.

Chapter 1

SCCS and its Descriptive Power

In this chapter, the formal language SCCS will be introduced and its expressiveness will be shown by proving SCCS to be equivalent with a Turing Machine. The 'elegance' of SCCS will be investigated by comparing its very important subset, CCS, with the comparable languages: CSP and Occam.

The relation between SCCS and CCS is sketched in the appendix. CCS is also handled here because it too can describe a Turing Machine. It also forms the basis of very fruitful research conducted under Prof. Koomen, by his graduates and promovendi. The transformation process developed for SCCS can serve as a model for an analogous transformation process for CCS. This point, however, is beyond the scope of this paper. It is an interesting topic for future work.

1.1 An introduction to SCCS

In this section, SCCS will be introduced. The introduction will begin with the presentation of its syntax and semantics. This will be followed by the presentation of the operational semantics of SCCS, given by Transition Graphs. The Transition Graph model will form the link between SCCS and the structure of finite-state machines. Transition Graphs can express the meaning of both an SCCS expression as that of a structure of finite-state machines.

The general discussion of SCCS will lead to the postulation of a normal form; a standard equational form in which all SCCS expressions can be written. Addressing normal form expressions, is semantically equivalent to addressing all SCCS expressions. The existence of a normal form is essential for the development of a transformation process to the CFSM model.

The definition of a Normal Form will be followed by a discussion of the communication model in SCCS. The communication model is perhaps the most important element in SCCS. It gives the ability to reason about interacting systems. The existence of this model makes a transformation to a structure of finite-state machines possible.

Before introducing SCCS, first some basic notions of the language itself. SCCS is a calculus which can be used to describe the behavior of synchronous systems. The behavior is described by asserting relations between the objects in the language. The objects are called 'Agent variables', or simply 'variables'. They can be thought of as a representation of 'states', as used in the automaton theory. The relations are symbolized by actions, which can be thought of as observable signals or events. A variable, 'P', can be given a meaning by asserting a successor relation, named by an action 'a', between it and another variable, P':

$P \xrightarrow{a} P'$

This relations can be read as: variable P is capable of performing action 'a', and in doing so, become P'.

The behavior described is implicitly synchronous. Time is discrete and its indivisible unit is the action. A successor relation $P \xrightarrow{a} P'$ implicitly means that variable P, existing at time 't', is capable of performing, in this time slot, the action 'a' and in doing so, become variable P' in '$t+1$'.

Now with the basic notion of the objects in SCCS, their relations and time, we turn to the formal definition of the language itself.

1.1.1 The Syntax

The syntax of SCCS is defined relative to two sets:

$\mathcal{V}$ar={A,B,...}

$\mathcal{A}$ct $= \{a, b, \ldots\}$

Both sets may be assumed to be inifinite. SCCS is defined by the following syntax free grammar, where 'E' will stand for an expression and 'I' is some set of indices, which can be thought of as some set of integers.

$$\mathrm{E} = a;E \mid \textstyle\sum_{i\in I} E_i \mid E\times E \mid E\restriction\{a,b,\ldots\} \mid$$
$$X \mid \mathrm{fix}_k\{X_i \mid i\in I\}\{E_i \mid i\in I\}, k\in I$$

1.1.2 The Semantics

(i) **Action:** '$a;$', $a \in \mathcal{A}\mathrm{ct}$, is a prefix operator. '$a;E$' is an expression with just one possible action and just one successor. It symbolizes the most basic binary relation:

$$a;E \xrightarrow{a} E$$

(ii) **Summation:** '$\sum_{i\in I} \tilde{E}$' is an expression. $\tilde{E} =< E_i \mid i\in I >$ is a, possibly infinite, indexed family of expressions. Summation is associative. It allows the expression of alternative actions.

Its *derivation* rule is:

$$\frac{E_i \xrightarrow{a} E' \quad (i\in I)}{\sum_{i\in I} E_i \xrightarrow{a} E'}$$

where $a \in \mathcal{A}$.

In a derivation rule, the validity of the relations above the line imply the validity of the relation below the line.

The behavior of a sum can be expressed by: $\sum_{i\in I} a_i;E_i \xrightarrow{a_i} E_i$.

Taking the sum over an index set, brings one to wonder about what it means to take the sum over an empty set. In an empty set of expressions, no actions are possible. The variable '$\mathbb{O}$' was chosen to symbolize an actionless expression. This variable is defined as:

$$\mathbb{O} \equiv \sum_{i\in\emptyset} \tilde{E}$$

For example, the expression 'a;$\mathbb{O}$' means that in the time slot following the action 'a', no more actions will occur.

By finite summation, another notation will often be used, namely:

$$E_1 + E_2 = \sum_{i \in \{1,2\}} \tilde{E}$$

In summary, summation is commutative, associative, with as identity element $\mathbf{0}$.

(iii) **Restriction** 'E$\restriction$A' is an expression and $A \in \mathcal{A}$. Its effect is to 'inhibit' expression 'E' from performing any actions in 'A'.

$$\frac{E \xrightarrow{a} E'}{E \restriction A \xrightarrow{a} E' \restriction A \quad (a \in A)}$$

The restriction operator has a number of important properties, here are a few of them:

1. $(\alpha; P) \restriction A \equiv \begin{cases} \alpha; (P \restriction A) & \text{If } \alpha \in A \\ \mathbf{0} & \text{If } \alpha \notin A \end{cases}$
2. $(\sum_i P_i) \restriction A \equiv \sum_i (P_i \restriction A)$
3. $P \restriction A \restriction B \equiv P \restriction (A \cap B)$

(iv) **Product:** '$\times$' is a binary operator. Its definition clearly reflects the notion of a sychronous time model. A synchronous time model supports the notion of 'simultaneity'. With the product operator, simultaneity can be expressed as follows:

$(a; E) \times (b; F) \xrightarrow{a \times b} E \times F$

Its derivation rule is:

$$\frac{E \xrightarrow{a} E' \qquad F \xrightarrow{b} F'}{E \times F \xrightarrow{a \times b} E' \times F'}$$

Milner backs the idea of simultaneity by hypothesizing that ($\mathcal{A}$ct , $\times$) is an Abelian Semigroup. It must be noted that the same symbol, $\times$, is both the operator in expressions as well as the operator in Actions. This is done to emphasize the semantic meaning of the $\times$ symbol, namely, that of synchronous actions.

The commutativity of the product operator between actions implies commutativity of the product operator between expressions. Hence,

$E \times F$ has the same meaning as $F \times E$, for any pair of expressions, E and F.

An important property of the product operator, is that it distributes over sum:

$(\sum_{i \in I} a_i; E_i) \times (b; F)$ has the same meaning as: $\sum_{i \in I} (a_i \times b; (E_i \times F))$

(v) **Variables:** The variables themselves have no meaning in SCCS. Their existence however, is very important. They form the basis for expression manipulation. Informally, they can be viewed as markers which may be substituted by an expression. They are literally variables which can be replaced by actions. The expression by which a variable is to be replaced can be given by either definition, 'simple' substitution or by recursion. Once a replacement is given for a variable, that variable is said to be 'bound' to the replacement. It then assumes the meaning of the expression to which it is bound. If a variable is not bound, it is said to be 'free', free for any replacement.

Binding a variable through definition is often informally done in derivation rules. For example, in $P \xrightarrow{a} P'$, the variable 'P' has apparently been bound to an expression which contains the subexpression $a; P'$. Formally, one is to first define $P \equiv a; P'$, and for this P, the mentioned action rule is valid.

When binding a variable through definition, it is important that the equation to which the variable is bound does not contain the variable itself. The definition would otherwise be meaningless; it would not be unique.

Binding variables via 'simple' substitution proceeds with the operator: $E[A/B]$ where E is an expression, A is a variable and B is a *free* variable. It is to be read as a command to replace all free occurrences of the variable B by the variable A. The command $E[\tilde{A}/\tilde{B}]$ is to replace all free occurrences of B_i by A_i. The command assumes that $\tilde{A}$ and $\tilde{B}$ were predefined as:

$\tilde{A} = \{A_i \mid i \in I\}$ and $\tilde{B} = \{B_i \mid i \in I\}$.

The third, and last mentioned methode of binding variables, is through recursion. Informally speaking, recursion defines a, possibly endless, systematic application of 'simple' substitutions. It is formally defined as follows:

(vi) **Recursion:** $\text{fix}_i\tilde{X}\tilde{E}$ is an expression, where $\tilde{E} =< E_i \mid i \in I >$ is an I-indexed family of expressions, $\tilde{X} =< X_i \mid i \in I >$ is an I-indexed family of distinct variables. The term '$\text{fix}\tilde{X}\tilde{E}$' is to be interpreted as a set of $\mid I \mid$ expressions: $\{\text{fix}_k\tilde{X}\tilde{E} \mid k \in I\}$.

The derivation rule for recursion is the following:

$$\frac{E_i[\text{fix}\tilde{X}\tilde{E}/\tilde{X}] \xrightarrow{a} E_i'}{\text{fix}_i\tilde{X}\tilde{E} \xrightarrow{a} E_i'}$$

The prefix, $\text{fix}\tilde{X}$, binds each variable X_i to expression E_i. The derivation shows that an expression, say $\text{fix}_k\tilde{X}\tilde{E}$, is to be interpreted as the expression E_k in which all occurrences of $\tilde{X}$ have been replaced by the expression to which they are bound. Thus, the expression '$\text{fix}_k\tilde{X}\tilde{E}$' has the same meaning as expression '$E_k[(\text{fix}\tilde{X}\tilde{E})/\tilde{X}]$'. It is clear that after a single substitution, each variable, 'X_i', in E_k is replaced by $\text{fix}_i\tilde{X}\tilde{E}$. In order to 'get rid' of the remaining recursion expressions in E_k, the substitution process must again be applied. The recursion operator thus defines a repeated application of 'simple' substitutions.

The substitution process may be finite, as the following example shows:

The expression:

$$\begin{aligned}&\text{fix}_1\{X_1, X_2\}\{\\ &\qquad E_1 \equiv a;\mathbb{O} + b;X_2\\ &\qquad E_2 \equiv c;\mathbb{O}\\ &\qquad\}\end{aligned}$$

has the same meaning as the following two expressions:

$a;\mathbb{O} + b;\text{fix}_2\tilde{X}\tilde{E}$

$a;\mathbb{O} + b;c;\mathbb{O}$

Note the use of formal definition, in the recursion expression, to make clear what the i^{th} expression is in the list of expressions $\tilde{E}$.

The substitution process may also be endless, as the following two examples show:

(i) $\text{fix}\{X\}\{a; X\}$ is semantically equivalent to 'a;a;$\text{fix}\{X\}\{a; X\}$'.

(ii)$\text{fix}\{X\}\{a; X \times X\}$ is semantically equivalent to:

a;$\text{fix}\{X\}\{a; X \times X\} \times \text{fix}\{X\}\{a; X \times X\}$

In general: a;a×a; a×a×a×a;....

The recursion expressions given above, described a SCCS expression in which the recursion term itself need not appear. The recursion term could be eliminated through substitution. In this case, the resulting expression is 'unique'. This is not the case for the following expression:

$\text{fix}\{X\}\{X + a; X\}$. This expression is has the same meaning as:

$\text{fix}\{X\}\{X + a; X\} + a;\text{fix}\{X\}\{X + a; X\}$

Because an original expression will never be prefixed by an action, no matter how many substitutions are made, the expression is not unique.

An important question is under which condition the recursion operation results in a unique expression. A sufficient condition, proven by Milner, is that every occurrence of $\tilde{X}$ in $\tilde{E}$ be prefixed by an action. A variable prefixed by an action is said to be 'guarded'. Before a recursion expression is transformed to its normal form, the existance of a unique solution will be checked.

It is important to add that with the recursion operator, it is possible to formally define the actionless variable, ⊘. It is defined as:

$\oslash \equiv \text{fix}\{X\}\{X\}$

In conclusion, we have seen that free variables can be bound to an expression by either 'simple' substitution, recursion or definition. This leads to the observation that the meaning of a SCCS expression is not

only represented by its possible actions, but also the actions which it may perform, obtained by a substitution of its free variables.

A special type of expression, is then an expression with no free variables. Such an expression has but one meaning. This type of expression is called an *Agent*. Agents form a special class, $\mathcal{P}$, of expressions within the language.

It is important now to present the equivalence relation used in SCCS. This will clarify what is meant by the meaning of an expression. The equivalence relation is first defined for Agents, and then extended to include expressions with free variables.

1.1.3 Strong Bisimulation

The equivalence defined by Milner is based on the notion of bisimulation.

Definition 1.1.1 Strong Bisimulation is a binary relation, $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ between Agents. Let $E, F \in \mathcal{P}$ and $a \in \mathcal{A}ct$.
Then $< E, F > \in \mathcal{R}$ iff

1. If $E \xrightarrow{a} E'$ then, for some $F' \in \mathcal{P}$,
   $F \xrightarrow{a} F'$ and $< E', F' > \in \mathcal{R}$.
2. If $F \xrightarrow{a} F'$ then, for some $E' \in \mathcal{P}$,
   $E \xrightarrow{a} E'$ and $< E', F' > \in \mathcal{R}$.

Definition 1.1.2 If E and F possess a bisimulation, we express this by stating $E \sim F$.

As proven by Milner in [2], '$\sim$', is an equivalence relation. He also proved that strong bisimulation is a congruence in the language. Important is the extension he had to make from Agents to general expressions, before the congruence proof. The extension is stated as follows:

Definition 1.1.3 Let $\tilde{X}$ be the set of all free variables in E and F. Then $E \sim F$ if $E[\tilde{P}/\tilde{X}] \sim F[\tilde{Q}/\tilde{X}]$ for any set of Agents $\tilde{P}$.

Therefore, the congruence of SCCS expressions is found by first transforming it into an Agent. If congruence holds under all possible transformations, then the expressions are congruent. It can be inferred, from this remark, that Agents form the semantic domain of the language. It is, however, impossible to give a finite representation for most Agents. For this reason, it is 'pleasurable' to have a model which can give a finite representation of most Agents. Such a model is a special kind of directed graph, a Transition Graph. In the context of this thesis, the Transition Graph model is adequate because a finite representation can be given for all Agents which are transformable to a CFSM. Furthermore, this model will lend itself for a better understanding of the normal form expression, which will be presented in the second following subsection.

1.1.4 The Transition Graph Model

The meaning of a SCCS expression can be expressed by a Transition Graph (TG). The graph may have an infinite number of arcs, each labelled by a symbol from the set $\mathcal{A}\mathrm{ct} = \{a, \ldots\}$, called *actions*. Its definition is as follows:

Definition 1.1.4 A Transition Graph is a 4-tuple $< Q, Ac, k, D >$ where Q is a nonempty, possibly infinite set of states, $Ac \in \mathcal{A}\mathrm{ct}$, is a possibly infinite set of actions, $k \in Q$, is the begin state and D, is an ordered 3-tuple: $Q \times Ac \times Q$.

The set D are the derivations. $< q_i, a, q_j > \in$D, means:
$q_i \xrightarrow{a} q_j$.
'q_i' is referred to as the 'current state' and 'q_j' as the 'next state'.

The operational semantics of SCCS will be given by the function:
$St : SCCS \rightarrow TG$

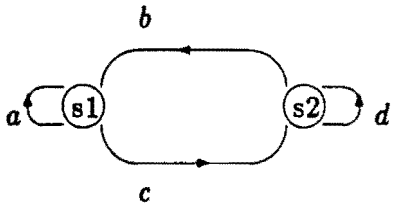
This function will be formally defined in subsection 1.2.1. This function defines a binary relation $\mathcal{R} \subseteq P \times Q$, between an Agent, say E, and a TG, say $< Q, Ac, k, D >$. This relation is such that:

1. $< E, k > \in \mathcal{R}$.
2. For every $E \xrightarrow{a} E'$, there is a $q \in Q$, such that $< k, a, q > \in D$, and $< E', q > \in \mathcal{R}$.

3. For every $< k, a, q > \in D$, there is a $E' \in P$, such that $E \xrightarrow{a} E'$ and $< E', q > \in \mathcal{R}$.

From this definition of St, we can find the operational meanings of Agents. For example, the expression '0', has as representation: $< \{k\}, \emptyset, k, \emptyset >$.

The following transition graph, in which the begin state is $s1$:



represents the SCCS expression:

$\text{fix}_1\{X_1, X_2\}\{$

$E_1 \equiv a; X_1 + c; X_2$

$E_2 \equiv b; X_1 + d; X_2$

$\}$

From the definition of St, it is obvious that the TG above represents an entire class of behaviors. Important is to find the minimum representation of a given class. To find this minimum, an equivalence relation is needed for Transition Graphs. As can be expected, the notion of bisimulation can also be formally defined for Transition Graphs.

The equivalence defined by Milner is based on the notion of bisimulation. For TGs, it can be easily defined. The bisimulation of graphs T1=$< Q_1, Ac_1, k_1, D_1 >$ and T2=$< Q_2, Ac_2, k_2, D_2 >$ is a relation R between their state sets, which contains their begin states. Let q_1 and q_1' be states in T1 and q_2 and q_2' be states in T2. The relation $R \subseteq Q_1 \times Q_2$, is defined that if

1. $< k_1, k_2 > \in R$.
2. $< q_1, q_2 > \in R$.
3. For every $q_1 \xrightarrow{a} q_1'$ in D_1 there exists a derivation $q_2 \xrightarrow{a} q_2'$ in D_2 for which holds: $< q_1', q_2' > \in R$.
4. For every $q_2 \xrightarrow{a} q_2'$ in D_2 there exists a derivation $q_1 \xrightarrow{a} q_1'$ in D_1 for which holds: $< q_1', q_2' > \in R$.

Definition 1.1.5 If T_1 and T_2 posses a bisimulation, we express this by stating $T_1 \sim T_2$.

Note that a bisimulation exists between a TG with an infinite number of states and a TG with only a finite number of states. The TG with the minimal number of states can represent the behavior of the members in the same equivelance class. A minimum TG can now be defined as follows:

Definition 1.1.6 A minimum Transition Graph, $< Q, Ac, k, D >$, is a Transition Graph for which the following two conditions hold:

1. For all $q, q' \in Q$, $q \neq q'$, holds: $< Q, Ac, q, D > \not\sim < Q, Ac, q, D >$.
2. All states are accessible from the begin state.

From the definition of St, it follows that all SCCS expressions can be represented by a TG. Given a TG, a minimal TG can be found. A normal form expression possesses similar characteristics. Every SCCS expression can be brought into normal form and a 'proper' normal form expression will be a 'minimum' representation.

The necessity of a normal form will become clear after the presentation of the CFSM model. The CFSM model is based on the structureless, finite-state machine model. In this model, a number of operators will be introduced. A finite-state machine can be directly translated to a type of transition graph. From this fact follows that, a SCCS expression, whose syntax allows for an easy translation to a Transition Graph, can represent a 'structureless' finite-state machine. Such an expression is a normal form expression, which is defined in the following subsection.

A normal form expression allows for a direct translation of finite-state machine characteristics to SCCS. Such a characteristic is the notion of 'input' and 'output'. This notion will be sketched in the section about the communication model in SCCS.

1.2 The Normal Form

In this section, the normal form will be presented. A normal form is essential for the transformation process from SCCS to the CFSM model. As mentioned in the previous section, a normal form will represent a 'structureless' finite-state machine. Later, we shall see that, a CFSM can be described by either a single normal form expression, or, by a product of normal form expressions.

The existence of a normal form implies that all SCCS expressions are provable, under bisimulation, to an expression in this form. As mentioned in the previous section, there will be a normal form and a 'proper' normal form. Given a normal form expression, an equivalent proper normal form can be found. The proper normal form will correspond to a 'minimal' representation of a behavior. Two propositions will be used to prove that for every expression, a normal form exists.

To aid the proof of the existence of a proper normal form, a number of transformation functions will be given. Together, the functions can transform any normal form expression to a proper normal form expression. It must also be proven that each transformation respects bisimulation. Hence, the expression before a transformation, must be congruent to the expression after a transformation. Here, three transformation functions will be given, each called an 'absorption lemma'.

In the description of the normal form, two functions, 'Succ' and 'Ungrd' are used. They are needed to describe an important characteristic of an expression. Both are a function from expressions to power sets of expressions. 'Succ(E)', short for 'successor', is to find all of the expressions which are related by an action, or a sequence of actions, with E. In terms of Transition Graphs, 'Succ' finds the set of nodes which are accessible from the node represented by E.

The function 'Ungrd(E)', short for 'unguarded', finds the set of free variables in expression 'E', which are not prefixed by an action.

Definition 1.2.1 Succ: Expression$\rightarrow \mathcal{P}$(expression)

$$
\begin{aligned}
\text{Succ}(\text{X}) &= \emptyset \text{ if X is free} \\
\text{Succ}(\textstyle\sum_i E_i) &= \cup_i \text{ Succ}(E_i) \\
\text{Succ}(a;E) &= \{E\} \cup \text{ Succ}(E)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Succ}(E \upharpoonright A) &= \mathrm{Succ}(E) \\
\mathrm{Succ}(\mathrm{fix}_k \tilde{X}\tilde{E}) &= \mathrm{Succ}(E_k[\mathrm{fix}\tilde{X}\tilde{E}/\tilde{X}]) \\
\mathrm{Succ}(E_1 \times E_2) &= \mathrm{Succ}(E_1) \times \mathrm{Succ}(E_2) \\
&\quad \text{where: } \{E_i \mid i \in I\} \times \{F_j \mid j \in J\} = \{E_i \times F_j \mid i \in I \wedge j \in J\} \\
&\quad \text{and } \{E\} \times \emptyset = \emptyset
\end{aligned}
$$

Definition 1.2.2 Ungrd: Expression$\rightarrow \mathcal{P}$(expression)

$$
\begin{aligned}
\mathrm{Ungrd}(X) &= \{X\} \text{ if X is free.} \\
\mathrm{Ungrd}(a;E) &= \emptyset \\
\mathrm{Ungrd}(\textstyle\sum E_i) &= \cup_i \mathrm{Ungrd}(E_i) \\
\mathrm{Ungrd}(E_1 \times E_2) &= \mathrm{Ungrd}(E_1) \cup \mathrm{Ungrd}(E_2) \\
\mathrm{Ungrd}(E \upharpoonright A) &= \mathrm{Ungrd}(E) \\
\mathrm{Ungrd}(\mathrm{fix}_k \tilde{X}\tilde{E}) &= \mathrm{Ungrd}(E_k[\mathrm{fix}\tilde{X}\tilde{E}/\tilde{X}])
\end{aligned}
$$

Now for the definition of a proper normal form expression:

Definition 1.2.3 An expression is in proper normal form if the following holds:

1. The expression is of the form: $\mathrm{fix}_k\{X_i \mid i \in I\}\{E_i \mid i \in I\}, k \in I$ and has a unique solution.

2. There is a function $L : I \times I \rightarrow \mathcal{P}(\mathcal{A}\mathrm{ct})$

   such that:

$$E_k \equiv F_k + \sum_{j \in I} \sum_{a \in L(k,j)} a; X_j$$

   where F_k is the sum of all free, unguarded variables in E_k, and do not include any variables in $\tilde{X}$.

3. Conditions (1) and (2) hold for all expressions in $\mathrm{fix}\tilde{X}\tilde{E}$.

4. For all $i, j \in I, i \neq j$, holds: $\text{fix}_i \tilde{X}\tilde{E} \not\sim \text{fix}_j \tilde{X}\tilde{E}$.

5. $\{\text{fix}_i \tilde{X}\tilde{E} \mid i \in I\} \subseteq \text{Succ}(\text{fix}_k \tilde{X}\tilde{E})$

A term is in *normal form* if conditions 1 and 2 hold.

A normal form expression is therefore a special recursion expression, as given by conditions one and two. If all of the expressions in a recursion expression are in normal form, we have then a set of expressions which lends itself well for a transformation to a Transition Graph. Absorption Lemma #1 shows how all expressions can be put in normal form.

It is very 'pleasant' to have a proper normal form which, when translated to a TG, will be a 'minimal' representation of a behavior in its equivalence class. For this reason, conditions four and five where added, which originate from the automaton theory. Condition four requires that all states in its TG transform be accessible and condition five requires that no two states belong to the same equivalence class. Absorption lemmas #2 and #3, shows how such a minimum can be found.

Absorption Lemma #1 The number of normal form expressions in $\{\text{fix}_i \tilde{X}\tilde{E} \mid i \in I\}$ can always be increased by one, unless, ofcourse, they are all in normal form. Given is therefore, a recursion expression with a unique solution, with $N \subset I$ expressions in normal form.

Let $\text{fix}_i \tilde{X}\tilde{E} \xrightarrow{a} \text{fix}_n \tilde{X}\tilde{E}$
where: $i \in N \wedge n \in I \wedge n \notin N$

1)For each $X_i \in \text{Ungrd}(E_n)$
DO
$E_n := E_n[E_i/X_i]$
OD.

2) For each $E_n \xrightarrow{a} E'$
DO
IF $E' \notin \{X_i \mid i \in I\}$
THEN
Let $l \notin I$
$I := I \cup \{l\}$

$E_l := E'$
$E_n := E_n[X_l/E']$
FI
OD
3) FOR $i \in I$
DO
$L(n,i) = \{a \mid a \in \mathcal{A}ct\ , E_n \xrightarrow{a} X_i\}$
OD

Correctness Proof:

The proof is given in three phases. First step (1) of the lemma is proven to respect congruence, and then the same is done for steps (2) and (3).

The first step does nothing more than to use the definition of recursion. Each unguarded variable is replaced by the expression to which it is bound by the fix$\tilde{X}$ prefix. Hence, the algorithm used in (1) repects bisimulation. Furthermore, Milner proved that, if the solution of a recursion expression is unique, then, after a finite number of substitutions, $E_k[\tilde{E}/\tilde{X}]$, each expression will be guarded in $\tilde{X}$. This implies that E_n will also be guarded in $\tilde{X}$ after a finite number of substitutions of the form $E_n[E_i/X_i]$. Hence, the algorthm used in (1) is also finite.

The second step is more complex. Let the resulting E_n after step one be given by:

$G_n \equiv \sum a; X_i + \sum a; P_j + F_n$

where $X_i \notin \{P_j\}$ for all $i \in I$ and F_n is the sum of all free, unguarded variables in E_n.

The algorithm in (2), takes each P_j, and binds a new X_l to it. Hence, after the first iteration of the loop,

$G_n \sim E_n[P_j/X_l]$

After the m^{th} iteration, m new X_l's, have replaced m P_j's. So that after m substitutions holds:

$G_n \sim E_n[\tilde{P}_j/\tilde{X}_l]$

From this follows that also step (2) respects bisimulation. This loop, given in this step, is certainly finite if the number of P_j's are finite.

The extension of the function L, in part (3), clearly does not effect the behavior of E_n. It is merely a decription of its behavior. This completes

the proof.

Here are two examples, in which a family of expressions is transformed into a family of normal form expressions.

$$\begin{aligned}\mathrm{fix}_2\{X_1, X_2\}\ \{&\\ &E_1 \equiv a; b; X_2 + e; \mathbb{O}\\ &E_2 \equiv c; X_1 + F\\ &\}\end{aligned}$$

is equivalent to the following proper normal form expression:

$$\begin{aligned}\mathrm{fix}_2\{X_1, X_2, X_3, X_4\}\ \{&\\ &E_1 \equiv a; X_3 + e; X_4\\ &E_2 \equiv c; X_1 + F\\ &E_3 \equiv b; X_2\\ &E_4 \equiv \mathbb{O}\\ &\}\end{aligned}$$

The function $L : I \times I \rightarrow \mathcal{P}(a, b, c, e)$, $I = \{1, 2, 3, 4\}$ of the proper normal form expression, can be given by one by a four vectors:

$L(1, I) = [\emptyset\ \emptyset\ \{a\}\ \{e\}]$
$L(2, I) = [\{c\}\ \emptyset\ \emptyset\ \emptyset]$
$L(3, I) = [\emptyset\ \{b\}\ \emptyset\ \emptyset]$
$L(4, I) = [\emptyset\ \emptyset\ \emptyset\ \emptyset]$

The next example is of a behavoir, whose proper normal form is an infinite set of expressions.

$\mathrm{fix}_1\{X_1\}\{a; X_1 \times X_1\}$ is equivalent to the proper normal form expression:

$\mathrm{fix}_1\{X_i \mid i \in \mathcal{N} - \{0\}\}\{E_i \equiv a^{2i}; X_{i+1} \mid i \in \mathcal{N} - \{0\}\}$
where a^i is a product of 'i' a's.
As for the function $L : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{P}(\mathcal{A}\mathrm{ct})$, can be specified as follows:

$$L(i, j) = \begin{cases} a^{2i} & \text{if } j = i + 1 \\ \emptyset & \textit{otherwise} \end{cases}$$

Absorption Lemma #2 Reduction of the number of necessary expressions. Given are a family of expressions, some of which are in normal form. $\{\mathrm{fix}_i \tilde{X}\tilde{E} \mid i \in I\}$

If $\text{fix}_l\tilde{X}\tilde{E} \sim \text{fix}_j\tilde{X}\tilde{E}$, $l, j \in I$, then a smaller family of expressions can be constructed as follows:

1) For all $E_i \in \tilde{E}, E_i := E_i[X_j/X_l]$

2) reduce the family of expressions: $I' \equiv I - \{l\}$

In terms of Transition Graphs, this lemma simply state that if two states are congruent, then, a state can be eliminated by giving both states the same name.

Correctness proof:

It is very straight forward to prove that

$\text{fix}_k\{X_i \mid i \in I\}\{E_i \mid i \in I\} \sim \text{fix}_k\{X_i \mid i \in I - \{l\}\}\ \{E_i[X_j/X_l] \mid i \in I - \{l\}\}$

Because $\text{fix}_l\tilde{X}\tilde{E} \sim \text{fix}_j\tilde{X}\tilde{E}$, it follows that for all $i \in I$,

$E_k[\text{fix}_j\tilde{X}\tilde{E}/X_j] \sim$

$E_k[\text{fix}_l\tilde{X}\tilde{E}/X_j]$

It does not matter whether E_j or E_l is filled in for X_j. Therefore, X_j can replace X_l and X_l and E_l can both be eliminated.

Absorption Lemma #3 Elimination of unreachable expressions. Given is a recursion expression:

$\text{fix}_k\{X_i \mid i \in I\}\{E_i \mid i \in I\}$

Let Extra $= \{i \mid i \in I \wedge \text{fix}_i\tilde{X}\tilde{E} \notin \text{Succ}(\text{fix}_k\tilde{X}\tilde{E})$

If the set 'Extra' is not empty, then, a smaller family of expressions can be formed by simply eliminating all expressions in this set: $I' \equiv I - \text{Extra}$

In terms of Transition Graphs, this lemma simply states that all states which are not accessible from the begin state, can be eliminated.

Correctness Proof:

It is also very straight forward to prove that

$Q \equiv \text{fix}_k\{X_i \mid i \in I\}\{E_i \mid i \in I\} \sim$

$\text{fix}_k\{X_i \mid i \in I - \text{Extra}\}\{E_i[X_j/X_l] \mid i \in I - \text{Extra}\}$

Because the expression in 'Extra' are 'unreachable', they will never be used in the substitution process:

$E_k[\text{fix}\tilde{X}\tilde{E}/\tilde{X}]$

If it were, then it is a successor of E_k, and which is a contradiction.

Proposition 1.2.1 *Every expression may be proven equal, up to bisimulation, to a normal form.*

This proposition will need two other propositions to prove. The proof can proceed by induction. Given a normal form expression, conditions (4) and (5) of definition 1.2.3 can be reached through absorption lemmas 2 and 3. In a family of expressions, the number of expressions in normal form can always be increased, as proven in absorption lemma 1. Hence, every normal form expression can be brought into proper normal form.

The next proposition will show that every expression can be placed in the recursion form, and the last proposition shows that a normal form exists for every expression in recursion form. This will conclude the proof.

Proposition 1.2.2 *Every expression, E, which is not in recursion form is congruent to an expression in recursion form.*

proof:

Let X be a variable for which holds: $X \notin \mathrm{Succ}(E)$ and $X \notin \mathrm{Ungrd}(E)$.

Then: $E \sim \mathrm{fix}\{X\}\{E\}$.

end of proof.

Proposition 1.2.3 *Every recursion expression with a unique solution is congruent to a normal form.*

proof:

If the expression $\mathrm{fix}_k\tilde{X}\tilde{E}$ has a unique solution, then absorption lemma 1 can be applied. When using this lemma, the set of expressions in normal form is empty, so that the first expression to be put in normal form is to be expression 'k', the 'begin expression'.

end of proof.

1.2.1 The Formal Relation Between SCCS and TGs

In this section the functions 'St' and 'Ts' will be formally defined. After the definition, it will be proven that the composition St(Ts()) is injective and that composition Ts(St()) is not. This should come as no surprise, since free variables have no meaning in TGs. This is because the result of the function 'Ts' is always an Agent.

Definition 1.2.4 St: $\{\text{fix}_k\{X_i \mid i \in Q\}\{E_i \mid i \in Q\}\} \rightarrow \{< Q, Ac, k, D >\}$

Let the SCCS expression be in proper normal form Hence, the begin state, k, and state space, Q, of the TG are already defined. Now for the set of derivations. Because the SCCS expression is in proper normal form, their exists an $L : Q \times Q \rightarrow Ac$

$$D \equiv \{< q, a, q' > \mid q', q \in Q \wedge a \in L(q, q')\}$$

we immediately see that the behavior of the SCCS expression is fully captured iff it is in proper normal form and all free variables in each expression are congruent to nil.

Definition 1.2.5 Ts: $\{< Q, Ac, k, D >\} \rightarrow \{\text{fix}_k\{X_i \mid i \in Q\}\{E_i \mid i \in Q\}\}$

$\text{fix}_k\{X_i \mid i \in Q\}\{E_i \mid i \in Q\}$ is an expression in proper normal form, in which 'L', $L : Q \times Q \rightarrow Ac$, is defined as:

For each $q, q' \in Q$

$L(q, q') = \{a \mid < q, a, q' > \in D\}$

so that, for all $q \in Q$

$$E_i \equiv \sum_{j \in I} \sum_{a \in L(i,j)} a; X_j$$

proof: $St(Ts(\{TG\})) = \{TG\}$

the proof is very straight foward. 'St' is injective for expressions in proper normal form with no free variables, and 'Ts' is injective for all TGs. This implies that $St(Ts(\{TG\})) = \{TG\}$

proof: $Ts(St(\{SCCS\})) \neq \{SCCS\}$

This follows directly from the fact that 'St' is not injective for expressions with free variables.

1.2.2 Communication in SCCS

In this subsection, the communication model in SCCS will be given. It will also be extended with important ideas which resulted from the transformation to the CFSM model. In chapter 3, the notion of communication in SCCS will again be revisited, but then more formally.

The communication model in SCCS is based on the group theory. Milner applies this theory by postulating that $(\mathcal{A}\text{ct}, \times, 1)$, is a group. With the characteristics of a group as basis, it is possible to postulate the existence of two kinds of actions: 'external' and 'internal'. External actions are 'observable'. They are communication actions which are accepted, or performed, by the observer. Internal actions are 'unobservable'. They correspond to communication actions between Agents in a network. The action '1' is an internal action. All other actions in $\mathcal{A}\text{ct}$ are 'external'.

Before this section, $(\mathcal{A}\text{ct}, \times)$, was only a semigroup. Composing expressions is then nothing more than a description of their 'concurrent' behavior. The external actions of each Agent, remained external in a product of expressions. No actions were considered to be communication actions between Agents within a network.

With the group theory as basis, a designer can compose a system with a very important fact in mind: For every action in a group, an inverse action exists. Their product is identity; an 'internal' action.

When composing a system from individual expressions, the designer will usually divide the actions, performable by each Agent, into one of two catagories: actions whose inverse is present within the system and actions whose inverse will be produced by the observer.

Actions in the first category can be grouped into pairs. The product of each pair is the identity action. The product of all combinations of these pairs are 'internal' actions; they are products of the identity action.

All possible combinations of actions in the second category are the external actions. Because the inverse of these actions are 'held' by the observer, the action '1' will not occur in the external action set.

A designer then composes the system by taking the product of the Agents in the network and projecting all actions on the set of all external and internal actions:

(product) $\restriction$ {external actions}$\cup${1}

For example,let $Q \equiv c;Q' + b;Q''$, $E \equiv b;E' + 1;E''$, $F \equiv a;F' + d;F''$.
Let the 'internal actions' be: $\{a \times b, 1\}$, where $a \times b = 1$
Let the 'external actions' be: $\{c, d, c \times d\}$

The network is then:
$(c;Q' + b;Q'') \times (b;E' + 1;E'') \times (a;F' + d;F'') \upharpoonright \{c, d, c \times d\} \cup \{1\}$
This expression is equivalent to:
$c;Q' \times E' \times F' \upharpoonright \{c, d, c \times d, 1\} + c \times d;Q' \times E'' \times F'' \upharpoonright \{c, d, c \times d, 1\} +$
$1;Q'' \times E'' \times F' \upharpoonright \{c, d, c \times d, 1\}$

Notice that if all actions were external actions, then their product would consist of six terms. This would be a description of their 'concurrent composition'.

This communication model is very basic, nothing is said about more complex notions, such as, channels, input or output. For their introduction in SCCS, they have to be expressed in terms of the communication model. Here, only the notion of 'channels' will be informally presented. The introduction of input and output will be saved for Chapter 3. The formulation of channels is made possible by the transfromation process given in Chapter 3.

The notion of channels can be expressed in terms of the model as the existence of a set of actions, which form a group, and are shared by only a pair of Agents within a network. In the example above, the pair of internal actions, (a, b), do not form a channel shared by Agents E and F, because the action 'b' can also be performed by Agent Q.

Milner offers a convenient methode to deal with channels. The idea is to presuppose a set of particles,Λ, which can generate all actions, internal and external, in a network. Each action performed by an Agent can then be characterized by their sort; the smallest subset of Λ needed to generate all of the actions which can be performed by an Agent and its successors. This is the set of actions in the Agent's Transition Graph.

With the existence of particles, the presents of a channel can now be more precisely defined. A channel exists between two Agents in a network, if they are each the sole possessor of one of two particles, which together generate a group.

For example, let $\{a_1, a_{-1}, b_1\} = \Lambda$, and $((a_1, a_{-1})^*, *, 1)$ is a group. Let:

$A \equiv a_1; A_1 + a_1^2; A_2$
$B \equiv b_1 \times a_{-1}; B_1$
then Sort(A)=$\{a_1\}$, Sort(B)=$\{b_1, a_{-1}\}$

The external actions are: $\{(b_1)^* - \{1\}\}$. Then internal actions are the infinite set of pairs: $\{(a_1^i, a_{-1}^i) \mid i \in \mathcal{N}\}$.

The system is then described by the product: $A \times B \restriction (b_1)^*$

Notice that the projection set could have also been defined as:

$(\Lambda - \{a_1, a_{-1}\})^*$. Milner gives a handier operator to specify this projection set:

Definition 1.2.6 The Restriction operator '$\backslash A$', $A \subset \Lambda$, is defined as:
$\restriction(\Lambda - A)^* \equiv \backslash A$

The restriction operation eliminates, in effect, all expressions which are prefixed by an action which contains a particle in A.

In the previous example, the restriction set is $\{a_1, a_{-1}\}$. The difference with the projection set is that this set is independent of the size of the larger set of particles to which they may belong. In the next section, which comes after the next subsection, the advantage of the restriction operator will become clear.

1.3 Turing Machine Equivalence

In this section, it will be shown that the expressive power of SCCS and a TM (Turing Machine) are the same. This will be achieved by first showing that a TM can express everything what SCCS can and then by showing that SCCS can describe a TM.

The proof that a TM can express everything SCCS can is very straight forward. It can best be done with Church's hypothesis, which reads that any process which can naturally be called a procedure, can be encoded in a TM.

SCCS can be viewed as nothing more than a set symbols with syntax rules and a number of operators which define string manipulation. Because all of the operators are procedurally described, they can all be encoded in a TM. Thus, all manipulations, which can be performed by SCCS can be performed by a TM.

Now for the proof that SCCS can describe a TM. The description of a TM will proceed in two steps. First it will be shown that SCCS can describe a push-down store automaton. With the push-down store, SCCS will describe a TM.

1.3.1 The Push-Down Store

The push-down store will have one input and one output alphabet. The input alphabet has particles: $\{req?, w?x \mid x \in X\}$. The action set '$w?x$', is a request to write the symbol 'x' to the stack. The action 'req?' is a request to read the last symbol to be put on the stack. The output alphabet has particles: $\{r!x \in X\}$. The action '$r!x$' will come in reponse to a read request. The data 'x' becomes available by this action.

An important symbol in X is the 'space' symbol: #. It is a symbol which the output action will produce if the stack is empty. However, because this symbol is in X, it may also be placed on the stack. This implies that if one wishes to detect whether the stack is empty, a restriction must be made on the use of this symbol.

The specification of a pushdown store automaton is the following:
Let $x, v, b \in X$

$PD(\$) = \sum_{x \in X} w?x; PD(x.\$) + req?; 1; 1; r!\#; PD(\$)$
$PD(v.R) = \sum_{x \in X} w?x; PD(x.v.R) + req?; 1; 1; r!v; PD(R)$

The string '$x.v.R$' symbolizes the symbols on the stack. The left most symbol is the symbol on the top of the stack. 'R' represents any string of symbols: $R \in X^*$.

Notice the use of the 'silent actions' in the specification. This is due to the 'delay' time between the request of data and its presentation. In SCCS, the time between actions is important.

In the following specification, there will be a set of agents which are dynamically created. This set can be defined by the following infinite set of recursive expressions:

For each $x \in X$

$S_i!x \equiv \text{fix}_i \tilde{X}\{E_i \equiv req_i?; s_i!x;\ 1 + 1; X_i \mid i \in \mathcal{N}\}$

Each agent, $S_i!x$, can informally be viewed as the i^{th} stack element, which contains the symbol 'x'.

The realization in SCCS of this specification is the following:

Let $I = \mathcal{N} - \{0\}$

$P_k \equiv \text{fix}_k\{Y_0, Y_i \mid i \in I\}$
$\{E_0 = \sum_{x \in X} w?x; (Y_1 \times S_1!x) \setminus \{s_1\} + req?; 1; 1; r!\#; Y_0,$

$E_i = \sum_{x \in X} w?x; (Y_{i+1} \times S_{i+1}!x) \setminus \{s_{i+1}\} + req?; req_i!; \sum_{x \in X} s_i?x; r!x; Y_{i-1}$
$\mid i \in I\}, k \in I$

Restriction over 's_i', is meant restriction over the set of particles: $\{s_i?x, s_i!x \mid x \in X\}$.

The proof of the realization is straight foward. The proof will be by induction.

$PD(\$) = P_0$, and
$PD(v.R) = (P_i \times S_i!v \times S_{i-1}!a \times \ldots \times S_1!b) \setminus \{s_1, \ldots, s_i\})$

such that $v.R = v.a \ldots b$

Induction step:

$$(P_i \times S_i!v \times S_{i-1}!a \times \ldots \times S_1!b) \setminus \{s_1, \ldots, s_i\}$$

$\approx$ {def of P_i}

$$(\sum_{x \in X} w?x; (P_{i+1} \times S_{i+1}!x)) \times (S_i!v \times S_{i-1}!a \times \ldots \times S_1!b) \setminus \{s_1, \ldots, s_i, s_{i+1}\}$$
$$+req?; req_i!; (\sum_{x \in X} s_i?x; r!x; Y_{i-1}) \times (S_i!v \times S_{i-1}!a \times \ldots \times S_1!b) \setminus \{s_1, \ldots, s_i\}$$

$\approx$ {def:$(S_i!v \times req_i; \sum s_i?x; E_i) \approx 1; 1; E_i$}

$$\sum_{x \in X} w?x; (P_{i+1} \times S_{i+1}!x \times S_i!v \times S_{i-1}!a \times \ldots \times S_1!b) \setminus \{s_1, \ldots, s_i, s_{i+1}\}$$
$$+req?; 1; 1; r!v; (P_{i-1} \times S_{i-1}!a \times \ldots \times S_1!b) \setminus \{s_1, \ldots, s_{i-1}\}$$

{with the induction hypothesis}

$$PD(v.R) = \sum_{x \in X} w?x; PD(x.v.R) + req?; 1; 1; r!v; PD(R)$$

And with the same technique, it is very easy to show:

$$PD(\$) = \sum_{x \in X} w?x; PD(x.\$) + req?; 1; 1; r!\#; PD(\$)$$

And that concludes the correctness proof.

1.3.2 The Turing Machine

A Turing Machine consists of a finite control, a tape and a head, which can read or write on the tape and move to the left or the right. The tape itself is infinite, it can hold an infinite number of symbols. The formal definition of a TM(Turing Machine), is the following:

A Turing Machine is a 4-tuple:(Q, X, k, δ)

- Q: is a finite set of state.
- X: is a finite alphabet with a special symbol: '#' the 'blank'.
- $k \in Q$ is the begin state.
- $\delta : Q \times X \rightarrow Q \times (X \cup \{L, R\})$

  $L, R \notin X$, are special 'command symbols' for the 'tape head'.

In the begin state, the tape is assumed to contain a finite number of non-blank symbols in the direct vicinity of the head. These symbols serve as the input of the TM. The TM operates as follows: It first reads a symbol currently under the head. The TM then processes this symbol with its function δ, which produces the next state and sends a symbol,'z' to the head. The head must respond correctly to the sent symbol. The response will be formally defined shortly. Informally, the response is as follows: If $z \in X$, the symbol 'z' must be written on the tape. If $z = L$, the tape must be moved one space to the left. If $z = R$, the tape must be moved one space to the right.

The operation of a TM can be formalized by specifing the current state, the contents of the tape to the left, right and under the head. The current status will be given before the head reads the contents of the tape. The current status will be referred to as the 'configuration'.

The definition of 'configuration' depends on the tape being used. Here, the tape will be viewed as two push-down stores.

Definition 1.3.1 *The configuration is a member of:*
$Q \times \$.X^* \times X \times X^*.\$$, *where* $\$ \notin X$

Thus, $(q, \$.a.b, c, d.\$)$ and $(q, \$, \#, a.b.\$)$ are configurations, but, $(q, a, b, c.\$)$ is not. The last three arguments of the configuration, can be concatenated to form the entire contents of the tape: $(q, \$.a.b, c, d.\$) \Rightarrow \$.a.b.c.d.\$$. The '$' is the 'empty' sign of a stack. In terms of an infinite tape, it is to be interpreted as an infinite string of blank symbols.

The head response can now be formally defined as follows:

Let $(q, A.a, b, c.C)$ be a configuration. Then, if $\delta(q, b) = (q', z)$, $z \in X \cup \{L, R\}$, the next configuration can be formally specified as follows:

Start: $(q, A.a, b, c.C)$ and $\delta(q, b) = (q', z)$

If $z \in X$ Then $\vdash (q', A.a, z, c.C)$

If $z = L$ Then If $A.a \neq \$$ Then $\vdash (q', A, a, b.c.C)$
Else $\vdash (q', \$, \#, b.c.C)$

If $z = R$ Then If $c.C \neq \$$ Then $\vdash (q', A.a.b.c, C)$

Else $\quad \vdash (q', A.a.b, \#, \$)$

Next, the specification of the finite controller will be given, followed by that of the head and tape. This subsection will then close with a sketch of the correctness proof.

First the finite controller. It is to read the symbol currently under the head and then, depending on its state, it is to send a symbol in $X \cup \{L, R\}$.

$$C_k \equiv \text{fix}_k\{Y_q \mid q \in Q\}$$
$$\{E_q \equiv req_T!; \textstyle\sum_{x \in X} r_T?x; w_T!z; Y_{q'}$$
$$\mid q \in Q\}, k \in Q$$

Where $\delta(x, q) = (z, q')$

The Head and Tape:

$$H_b \equiv \text{fix}_b\{Y_i \mid i \in X\}$$
$$\{E_i \equiv req_T?; r_T!i; Y_i$$
$$+ \textstyle\sum_{d \in X} w_T?d; Y_d$$
$$+ w_T?L; w_R!i; req_L!; \textstyle\sum_{d \in X} r_L?d; Y_d$$
$$+ w_T?R; w_L!i; req_R!; \textstyle\sum_{d \in X} r_R?d; Y_d$$
$$\mid i \in X\}, b \in X$$

And finally, the push-down stores:

$$PD_L(\$) = \textstyle\sum_{x \in X} w_L?x; PD(\$.x) + req_L?; 1; 1; r_L!\#; PD_L(\$)$$
$$PD_L(A.a) = \textstyle\sum_{x \in X} w_L?x; PD_L(A.a.x) + req_L?; 1; 1; r_L!v; PD_L(A)$$

$$PD_R(\$) = \textstyle\sum_{x \in X} w_R?x; PD_R(x.\$) + req_R?; 1; 1; r_R!\#; PD_R(\$)$$
$$PD_R(c.C) = \textstyle\sum_{x \in X} w_R?x; PD(x.c.C) + req_R?; 1; 1; r_R!c; PD_R(C)$$

The behavior expression of the Turing Machine is then:

$$\text{TM} = (C_q \times PD_L(A.a) \times H_b \times PD_R(c.C)) \backslash \text{Ch}$$

where 'Ch' is the set: $\{r_T, r_L, r_R, w_T, w_L, w_R, req_T, req_L, req_R\}$

The proof is very straight foward. It uses the same technique as the correctness proof of the push-down stores. All it does is to prove that, given a begin configuration and a particular value of 'z', the expected next configuration will result. Hence, the proof uses only simple rules in SCCS.

What must be proved is the following:

$$
\begin{aligned}
&\mathrm{TM} = (C_q \times PD_L(A.a) \times H_b \times PD_R(c.C)) \backslash \mathrm{Ch} \\
\approx\ &\{\text{if } z \in X\} \\
&1; (C_{q'} \times PD_L(A.a) \times H_z \times PD_R(c.C)) \backslash \mathrm{Ch} \\
\\
\approx\ &\{\text{if } z = L\} \\
&1; (C_{q'} \times PD_L(A) \times H_a \times PD_R(b.c.C)) \backslash \mathrm{Ch} \\
\\
\approx\ &\{\text{if } z = R\} \\
&1; (C_{q'} \times PD_L(A.a.b) \times H_c \times PD_R(C)) \backslash \mathrm{Ch}
\end{aligned}
$$

where $\approx$, is the sign of 'weak equivalence', defined in CCS, and means here that any number of '1' actions may pass before the final condition is reached.

Note that, no special case must be made if one of the two push-down stores becomes empty. This is because, if it is empty, the special symbol, #, will 'automatically' be produced. It should also be noted that the special symbol may also be placed on the stack This implies that its production says nothing about the number of elements still on the stack. This gives the stack its needed infinite character.

1.3.3 Conclusion

An interesting consequence of the equivalence is, is that the halting problem of the TM must have an analogue in SCCS.

The halting problem of the Turing Machine is simply the unsolvable problem of determining whether an arbitrary Turing Machine in an arbitrary configuration will ever halt. It is unsolvable because one can only determine whether the Turing Machine in such a case will halt by actually letting the Turing Machine communicate with the tape.

The question of halting can be rephrased in SCCS as the question of *deadlocking*. Deadlock occurs if, after an expansion, the resulting expression

contains the variable '$\mathfrak{O}$'. In the case of the TM, The TM will halt if the expression:

$(C_q \times PD_L(A.a) \times H_b \times PD_R(c.C))\backslash \mathrm{Ch}$

has '$\mathfrak{O}$' as a successor.

The detection of deadlock cannot be solved any differently than the halting problem. The product of the behavior expressions will have to be calculated in order to determine whether or not the Turing Machine will halt. Since the composition and restriction operators can be described by the Turing Machine, these operators could never find a solution which the Turing Machine could not find.

1.4 A Comparison of CCS and CSP

In this section the syntax and equivalences of CCS and CSP will be compared. This section and the next are the only two sections in which CCS plays a role. CCS is used here because it is an asynconous language, just as CSP and OCCAM. Because CCS can be derived from SCCS, the behaviors CCS describes can be viewed as a subset of behaviors describable by SCCS. The relation between SCCS and CCS is fully described by Milner in [2].

The comparison is based on two references, one is an article by S.D Brookes [6] and the other is a book published by C.A.R. Hoare [7].

1.4.1 Equivalences and Operators

In his article, Brookes gives a complete axiomatization of CSP, using Milner's synchronization trees as the semantic model. This semantic model can be viewed as a subset of the Transition Graph model, namely all graphs which do not contain any loops. This axiomatization allows the comparison of essential properties of CSP and CCS. Two properties are thoroughly discussed in the article, that of CCS's and CSP's operators and equivalences.

Note, in the following, the symbols 'T' and 'S' represent behavior trees. Furthermore, the symbol 'τ', is a nother notation of the identity action, '1'.

1.4.2 Failure Equivalences

Brookes begins by defining *failure equivalences* in terms of synchronization tree attributes. The given definition forms a complete axioma system for failure equivalence on finite trees. The axioms are divided into two groups. One group forms the basic laws of synchronization trees, referred to as (A1) ... (A4). These are the following:

$$
\begin{array}{lrcl}
(A1) & S+T & \equiv & T+S \\
(A2) & (S+T)+U & \equiv & S+(T+U) \\
(A2) & S+S & \equiv & S \\
(A4) & S+NIL & \equiv & S
\end{array}
$$

The other group of axioms, are specific to failure equivalences, referred to as (B1) ... (B4) and (R). They are the following:

$$
\begin{array}{lrcl}
(B1) & S + \tau T + U & \equiv & \tau(S+T) + \tau T + U \\
(B2) & \tau S & \equiv & S \\
(B3) & \mu S + \mu T + U & \equiv & \mu(\tau S + \tau T) + U \\
(B4) & \tau(\mu S + T) + \tau(\mu S' + T') & \equiv & \tau(\mu S + \mu S' + T) + \tau(\mu S + \mu S' + T')
\end{array}
$$

And the inference rule, (R):

$$
\frac{S \equiv T}{\mu S + U \equiv \mu T + U}
$$

where $\mu \in (\Lambda \cup \{\tau\})$

For comparison, the laws specific for observation equivalences are the following:

$$
\begin{array}{lrcl}
(M1) & S + \tau S + T & \approx & \tau S + T \\
(M2) & \tau S & \approx & S \\
(M3) & \mu S + \mu(\tau S + T) + U & \approx & \mu(\tau S + T) + U
\end{array}
$$

And the inference rule, (R):

$$
\frac{S \approx T}{\mu S + U \approx \mu T + U}
$$

Observation equivalence is a finer equivalence than failure equivalence since the axioms of observation equivalence can be derived from those of failure equivalence. It appears that failure equivalence respects the minimum requirement for ***deadlock equivalence***. Deadlock equivalence is an equivalence defined by *Ir.H. Mettes*, from the 'TUE'. Deadlock equivalence can be defined as follows:

> If two agents are deadlock equivalent, then the set of agents, with which each can enjoy deadlock free communication, are identical.

1.4.3 Transforming CSP Syntax

In his article, Brookes defines a mapping from CSP syntax to synchronization trees. A CSP process and its synchronization tree 'image', will have

the same failure sets. This leads to the assertion by Brooks that two CSP processes have the same meaning in the failure set semantics if and only if their synchronization tree images are equivalent.

For the mapping process, the function $\mathcal{T}$ is introduced. The function is defined, so that if P is a CSP process, $\mathcal{T}[\![P]\!]$ is its synchronization tree image. The following clauses define the map $\mathcal{T}$:

$$\begin{aligned}
\mathcal{T}[\![STOP]\!] &\equiv NIL \\
\mathcal{T}[\![a \rightarrow P]\!] &\equiv a : \mathcal{T}[\![P]\!] \\
\mathcal{T}[\![P_1 \sqcap P_2]\!] &\equiv \tau : \mathcal{T}[\![P_1]\!] + \tau : \mathcal{T}[\![P_2]\!] \\
\mathcal{T}[\![P_1 \Box P_2]\!] &\equiv \mathcal{T}[\![P_1]\!] \Box \mathcal{T}[\![P_2]\!] \\
\mathcal{T}[\![P_1 \| P_2]\!] &\equiv \mathcal{T}[\![P_1]\!] \| \mathcal{T}[\![P_2]\!] \\
\mathcal{T}[\![P_1 \|\!| P_2]\!] &\equiv \mathcal{T}[\![P_1]\!] \|\!| \mathcal{T}[\![P_2]\!] \\
\mathcal{T}[\![P/b]\!] &\equiv \mathcal{T}[\![P]\!][\tau b]
\end{aligned}$$

The three operators, $\Box, \| , \|\!|$, are defined as follows:

with:

$$S = \sum_{i=1}^{n} a_i : S_i + \sum_{i=1}^{N} \tau : S_i', \quad T = \sum_{j=1}^{m} b_j : T_i + \sum_{j=1}^{M} \tau : T_j'$$

we have:

$$S \Box T = \sum_{i=1}^{n} a_i : S_i + \sum_{j=1}^{m} b_j : T_j + \sum_{i=1}^{N} \tau : (S_i' \Box T) + \sum_{j=1}^{M} \tau : (S \Box T_j'$$

$$S \| T = \sum_{a_i = b_j} a_i : (S_i \| T_j) + \sum_{i=1}^{N} \tau : (S_i' \| T) + \sum_{j=1}^{M} \tau : (S \| T_j')$$

$$S \|\!| T = \sum_{i=1}^{n} a_i : (S_i \|\!| T) + \sum_{j=1}^{m} b_j : (S \|\!| T_j) + \sum_{i=1}^{N} \tau : (S_i' \|\!| T) + \sum_{j=1}^{M} \tau : (S \|\!| T_j')$$

The '$\|\!|$' operator closely resembles the '$|$' operator in CCS. It would yield the same result if $\mathcal{A}ct$ were assumed to be a monoid and not a group. That

CSP does not use group theory can also be seen by the definition of the '||' operator. It is 'strange' because the product of two actions which are to 'communicate', $a_i = b_j$, is to result in 'a_i' instead of 'τ'. For this reason, its cannot be correctly formulated in CSS. One has to first take a step back to SCCS. Instead of of projecting each expression on $\Lambda \cup \{1\}$, it must be projected on $\Lambda \cup \{1\} \cup \{a_i \times b_j \mid a_i = b_j\}$. After projection, we can rename the product terms by a_i. In summary:

$$S \| T \approx \Phi[S \times T \restriction \Lambda \cup \{1\} \cup \{a_i \times b_j \mid a_i = b_j\}]$$

where $\Phi[\{a_i \times b_j \mid a_i = b_j\}] = \{a_i\}$

1.4.4 A Comparison of Frameworks

Now that the syntax and equivalences of CCS and CSP can be expressed in a common semantic model, the framework of the two languages can be compared.

In his book, C.A.R Hoare informally compared the frameworks of CCS and CSP. The most interesting of his remarks concerned the identification properties of the two languages and the opertors used.

Hoare noted that CSP makes as many indentifications as possible, while preserving only essential distinctions. CCS, on the other hand, makes only essential identifications. His statements are backed by the findings in the previous section. CCS makes many, finer equivalences. CCS's strongest equivalence is identity, and its weakest is behavior equivalence. In contrast, Hoare defines a single, very weak equivalence: Failure equivalence. That *essential distinction*, mentioned by Hoare, is probably *deadlock equivalence*.

While Hoare also stated that CCS strives for maximum expressiveness with a minimum of operators, CSP is settled with as many operators as appropriate. This statement is also backed by the findings of the previous section.

1.5 A Comparison of OCCAM and CCS

In this chapter, OCCAM's primitive processes and constructs will be translated into CCS. This will be followed by a brief discussion of the frameworks of the two languages. The syntax of OCCAM used is that defined in INMOS's 'OCCAM Programming Manual' [8].

OCCAM's syntax can be translated into CCS. The syntax is divided into seven categories: Processes, Primitive processes, Constructors, Declarations, Expressions, Vector operations and Configuration. Only the primitive processes and constructors will be handled here because they form the very hart of an OCCAM program.

1.5.1 OCCAM's Primitive Processes and Constructs

In this subsection, OCCAM's primitive processes and constructs will be translated into CCS.

1.5.2 Primitive Processes

OCCAM has basically three primitive processes: assignment, input and output. The in- and output processes are identical to CCS'S. Input is noted as:

<channel name> '?' <variable>

where 'variable' can either be a list of zero or more variables. The output is noted as:

<channel name> '!' <expression>

where 'expression' is simply any expression of zero or more variables.

It is important to realize that channels can only pass values and not variables. All variables are in the local memory. The assignment process of a value to a variable, can therefore be seen as a 'read' and 'write' action to a local register. The assignment :

<variable> ':=' <value> can be noted in CCS as:

$Reg_v! < value >$

In CCS, local register can be modelled by the following agent:

$\text{fix}_x\{V_x \mid x \in X\}$

$\{E_x \equiv Reg_v? < value >: V_{<value>} + reg? : Reg_v!x : V_x$

$|\ x \in X\}$

It is also assumed that $< value > \in X$.
An expression can therefore be translated in the following steps:

1. 'read' the variables used in the expression.
2. evaluate the expression
3. 'write' the resulting value to the variable to which the expression is assigned.

1.5.3 Constructs

OCCAM has five constructs: SEQ,PAR,ALT,IF and WHILE. The following table gives their translation into CCS:

OCCAM Constructs	CCS Equivalent
SEQ A B	A:B
PAR A B	A|B
ALT A B	A+B
WHIlE B P	$\text{fix}\{A\}\{r?B : (\sum_{B=True} P : A + \sum_{B=False} D)\}$
IF B P NOT B Q	$in?B : (\sum_{B=True} P + \sum_{B=False} Q)$

1.5.4 A Comparison of OCCAM's and CCS's Frameworks

OCCAM is, according to C.A.R. Hoare, based on a simple version of CSP. However, CSP and OCCAM are based on two different philosophies. In [9], R.Talyor, from INMOS Ltd, states that OCCAM creaters strived for a concurrent language with relatively few software constructs. This motivation is also reflected by the name of the language. The concept of 'Occam's razor' is a plea to keep the constructs simple. This, however, was not Hoare's goal. Hoare seeks simplicity in a model in which operators are easy to create. The idea of using few operators with a maximum of expressiveness is the idea behind CCS. In this sense, it seems that OCCAM has closer ties with CCS than CSP. A translation into CCS will be one in which OCCAM's idea of simplicity will be retained. Therefore, an OCCAM program can be formalized without losing either meaning nor expressiveness; OCCAM's 'razor' will remain untarnished.

Chapter 2

Communicating Finite State Machines

In this chapter, the underlying FSM model will be introduced. As mentioned in the introduction of this thesis, the CFSM (Communicating Finite State Machine) model must be compositional to express structure. The description of an element in the structure must be that of a 'standard' FSM. This will allow the use of standard implementation techniques for the generation of an actual circuit.

The model will be made compositional by the definition of a 'connect' operator with which individual components can be 'connected'. A component will be expressible as either a structure or by a 'standard' FSM specification. A standard FSM model will be chosen which is closed under the connect operation. Without the closure property, the set of all CFSMs would not be well-defined. A 'connect' operation may otherwise not yield a CFSM.

In the first section, a CFSM will be defined. It is a standard FSM model which will later be proven to be closed under the connect operator. The standard model used is that of a 'State-Output Machine', which is defined in the following section.

In the second section, the connect operation will be given. Its definition is based on a certain communication model. It is very important to be able to prove the models' consistancy. For this reason, the model chosen is the same as that given by Milner in SCCS. This has had as result, that the defined 'connect' operator can be proven to 'commute' with a composition

of operators in SCCS. This implies that the CFSM model is homomorphic with its SCCS description. This homomorphism will be shown in Chapter 3.

In section 2.3, the notion of bisimulation will be introduced in the CFSM model. This equivalence relation will be a congruence for the connnect operation. The relation is 'stronger' than 'string' equivalence, the equivalence used between finite-state machines.

2.1 The CFSM

This section will begin with the formal definition of a State-Output Machine and its semantics. Its semantics will be expressed by Transition Graphs, defined in Chapter 1. This will lead up to the definition of a CFSM.

2.1.1 The State-Output Machine

A SOM (State-Output Machine) is formally defined as a 6-tuple:$< Q, X, Y, k, p, m >$, where:

- Q is a finite set of states.
- X is the finite 'input alphabet'.
- Y is the finite 'output alphabet'.
- $k \in Q$ is a begin state.
- $p : Q \rightarrow Y$ is the 'output function'.
- $m : Q \times X \rightarrow P(Q)$ is the 'next state relation',

It is important to test whether a SOM is applicable in an actual circuit. In a 'real' circuit, a SOM is continuously 'doing' something, it is always either accepting or generating symbols. During normal operation, it should never enter a state in which no symbols are acceptable.

To test its applicability, we must first determine the set of accessible states, and then each of these states must have at least one next state.

The set of accessible states can be found as follows:

Let $S_1 = \{k\}$, the begin state. Then for $i \geq 1$:

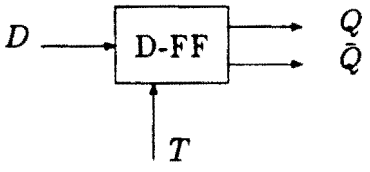
Let $P_i = \cup_{j \in \{1,\ldots,i\}} S_j$
$S_{i+1} = \{q \mid \exists a \in X : \exists s \in P_i : q \in m(s,x) \wedge q \notin P_i\}$

Because $S_{i+1} \cap S_i = \emptyset$ and if $S_i = \emptyset$ then $S_{i+1} = \emptyset$, it follows that:
Let $n = \mid Q \mid$
$\cup_{i=1}^{n} S_i = \cup_{i=1}^{\infty} S_i$ and
$\cup_{i=1}^{n} S_i \subseteq Q$

Now, the definition of an impelementable SOM can be given:

Definition 2.1.1 A SOM is *implementable* iff: $\{\forall_{q \in S} \exists_{a \in X} \mid m(q,a) \neq \emptyset\}$ where $S \subseteq Q$, is the set of reachable states.

In the model of a SOM, given here, both the input and output alphabets may be defined as the cartesian product of a finite number of sub-alphabets;
$X = X_1 \times \ldots \times X_n$ and
$Y = Y_1 \times \ldots \times Y_m$.
Considering the alphabet of a SOM as the cartesian product of sub-alphabets, has everything to do about the 'connect' operation, which is yet to be presented. The 'connect' operation will link-up a sub-input with a sub-output alphabet and will generate a CFSM in which these alphabets are 'eliminated' from the list of sub-alphabets; the sub-alphabets will be 'hidden'. This implies that a CFSM with only a single input and output alphabet, will 'dissappear' after a 'link-up'. Unconnected input and output alphabets serve, in effect, as observation points of the CFSM's behavior. Hence, the existence of multiple sub-alphabets gives a SOM the capacity to accept symbols from more than one generator and to generate different symbols for different acceptors.
In the physical model of a SOM, one prefers to speak about 'ports' instead of 'alphabets'. The physical model will have as many 'ports' as subalphabets. Hence, at each input port, a set of symbols, X, may be accepted and at each output port, a set of symbols, Y, may be generated.
Take the following D-FF for example:



Input: $D \times T$
Output: $Q \times \bar{Q}$

$D, T, Q, \bar{Q} = \{0, 1\}$

'D' is the data port, 'T' is the clock, 'Q' is the data output and '$\bar{Q}$' is the inverse of the data output. The function of a D-FF will be given in the next subsection. Important here is that we see that both the input and output alphabets are the cartesian product of two sub-alphabets. Here, each sub-alphabet corresponds to a physical port. Each sub-alphabet is the symbol set $\{0, 1\}$.

2.1.2 The Semantics of a SOM

The standard State-Output Machine model also consists of a standard equivalence relation to classify their behavior. This equivalence relation can best be described with the aid of Transition Graphs. This subsection will begin with the relation between TGs (Transition Graphs) and a SOM and end with a discription of the equivalence relation defined in the standard SOM model.

As mentioned, the meaning of a SOM can be expressed by a TG. The Transition Graphs to be used here, are defined relative to the set $U_x \times U_y$, where U_x is the set of all possible input symbols and U_y, the set of all possible output symbols. We have defined the input and output alphabets as expressible as the cartesian product of sub-alphabets. This means that configurations such as X or $X_1 \times \ldots \times X_n$, are all in the set U_x. An analogous trait holds for the set of all output actions, U_y. Important now, is to note what is not in $U_x \times U_y$. As mentioned in the previous subsection, we have presupposed that an input and output alphabet cannot be the cartesian product of an infinite number of sub-alphabets. They must remain recursively enumerable. Hence, such infinite configurations will not be in the set $U_x \times U_y$.

In the previous Chapter, the Transition Graph model was defined. Here, it will have a similar definition: a 4-tuple, $< Q, X \times Y, k, D >$, where Q is a *finite* set of states, k is the begin state and D is a *finite* set of derivations: $< q, (x, y), q' > \in D$, where $q, q' \in Q$ and $(x, y) \in X \times Y$.

Hence, the only difference in the TGs used here, is that they are finite and are defined relative to the set $U_x \times U_y$. In Chapter 3 the relationship between the set of pairs in $U_x \times U_y$ and the set of actions in $\mathcal{A}ct$, will be given.

In subsection 2.1.4, the transformation between a SOM and a TG is formally defined. Given this transformation, it can be derived which class of TGs can be transformed to a SOM. In the following subsection, it is proven that the following requirement must be met:

For each Transition Graph, there exists a function $\alpha : Q \rightarrow Y$. Such that:

$\{< q,(x,y),q' > \in D \Rightarrow y = \alpha(q)\}$

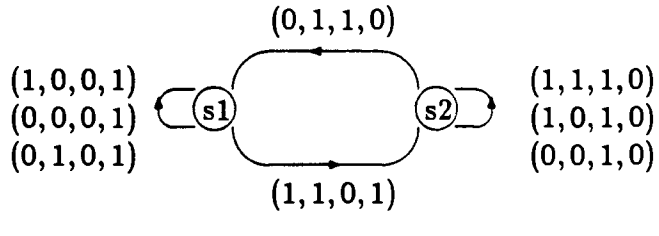
In words, each branch extending form a node, must have the same 'y' component in its label.

The TG is a representation of an implementable SOM if every reachable state in Q has at least one derivation:

$Q = \{q \mid < q,(x,y),q' > \in D\}$

Thus, if every reachable state can perform at least one action.

For example, the D-FF shown in the previous subsection, can be described by the following TG: The label set is: $D \times T \times Q \times \bar{Q}$ and state 's1' is the begin state.



'D' is the data port, 'T' is the clock, 'Q' is the data ouput and '$\bar{Q}$' is the inverse of the data output. The D-FF works as follows: it has two states, 's1' and 's2'. In 's1' the data-output is '1' and in state 's2', the data-output is '0'. If the clock receives a '0', then it remains in the same state. If the clock receives a '1', then the next state depends on the symbol received at the data-input. If that symbol is a '1', then the next state is 's2', if it is a '0', the next state is 's1'.

The meaning of a SOM can be expressed by specifying all of the *traces*

in a TG. A trace is a sequence of arc labels occuring in a path in a TG. The *trace set* of an SOM is the set of all traces beginning at the begin state. Two SOMs are equivalent if they have the same trace set. For example, in the graph above, 's1' is the begin state. There are four traces of length one, one of which is: (0,0,0,1). A trace of length two is: (0,0,0,1).(0,1,1,0).

This equivalence can now be compared to bisimulation, discussed in the previous chapter. The trace set defines a coarser classification of behaviors. At this point it can already be noticed that the trace set equivalence is too coarse for the SOMs defined here. The equivalence cannot differentiate an implementable SOM from an non-implementable one. This is because traces are unable to distinguish a deterministic SOM from a nondeterministic one. Take for example, the following two TGs:

$$\text{T1}=< \{q_1,q_2\},\{(x,y)\},q_1,\{< q_1,(x,y),q_1 >, < q_1,(x,y),q_2 >\} >$$
$$\text{T2}=< \{q_1\},\{(x,y),\},q_1,\{< q_1,(x,y),q_1 >\} >$$

They both have the same trace set, namely,
$\{\{(x,y)\},\{(x,y).(x,y)\},\{(x,y).(x,y).(x,y)\},\ldots\}$ But T1 is not implementable while T2 is. Hence the equivalence can allow the replacement of a implementable TG by a nonimplentable one. A finer equivalence is clearly needed. This equivalence will be based on the notion of bisimulation, and will be given in section 2.3.

2.1.3 The Communicating Finite-State Machine

A Communicating Finite-State Machine, can now be defined as a SOM. When working with a CFSM, it is easier to manipulate the function:

$B : Q \times Q \rightarrow P(X \times Y)$, rather than the next state and output functions. The function B defines a $\mid Q \mid$ square matrix, called the behavior matrix. In the rest of this subsection, it will be shown that a bijective function exists between the behavior matrix and the next state and output functions. From this relation it follows that the specification of a CFSM as $< Q, X, Y, k, p, m >$ is equivalent to the specification as $< Q, X, Y, k, B >$

For example, the D-FF, defined in terms of TGs, can be given the following CFSM definition: $< \{s1, s2\}, D \times T, Q \times \bar{Q}, s1, B >$, where B is:

	$s1$	$s2$
$s1$	$\{(1,0,0,1),$ $(0,0,0,1),$ $(0,1,0,1)\}$	$\{(1,1,0,1)\}$
$s2$	$\{(0,1,1,0)\}$	$\{(1,0,1,0),$ $(1,1,1,0),$ $(0,1,1,0)\}$

Definition 2.1.2 A behavior matrix of a CFSM, $< Q, X, Y, k, p, m >$, is defined by the function $B : Q \times Q \rightarrow P(X \times Y)$. This function is divisible into two functions:

$T : Q \times Q \rightarrow P(X)$ and $p : Q \rightarrow Y$.

such that, for all $q, q' \in Q$:

$B(q, q') = T(q, q') \times \{p(Q)\}$

The behavior matrix is therefore the composition of two functions, 'T' and the state output function 'p'. The function T defines a 'Transition Table'. They are often confronted in the literature. There exists a bijective function between a Transition Table and the next state function. Given this relation, and the definition of the behavior matrix, follows the existence of a a bijective relation between the behavior matrix and the state output and next state functions. With this bijective transformation, we can work with a CFSM as either the 6-tuple: $< Q, X, Y, k, p, m >$ or the 5-tuple: $< Q, X, Y, k, p, m >$.

In the following it will be proven that a bijective relation exists between a Transition Table and the next state function of a CFSM.

Definition 2.1.3 $Nt(< Q, X, m >) =< Q, X, T >$ where

$T : Q \times Q \rightarrow P(X)$ and $m : Q \times X \rightarrow P(Q)$

For each $q, q' \in Q$:

$T(q, q') = \{x \mid x \in X \wedge q' \in m(q, x)\}$

Definition 2.1.4 $Tn(< Q, X, T >) =< Q, X, m >$

$T : Q \times Q \rightarrow P(X)$ and $m : Q \times X \rightarrow P(Q)$

For each $x \in X$ and $q \in Q$:

$m(q, x) = \{q' \mid q' \in Q \wedge x \in T(q, q')\}$

proof: $Nt(Tn(< Q, X, T >)) =< Q, X, T >$
For each $q, q' \in Q$:
$T(q, q')$
= {definition}
$\{x \mid x \in X \wedge q' \in \{q' \mid q' \in Q \wedge x \in T(q, q')\}\}$
= {calculus}
$T(q, q')$
end proof.

proof: $Tn(Nt(< Q, X, m >)) =< Q, X, m >$
For each $x \in X$ and $q \in Q$:
$m(q, x)$
= {definition}
$\{q' \mid q' \in Q \wedge x \in \{x \mid x \in X \wedge q' \in m(q, x)\}\}$
= {calculus}
$m(q, x)$
end proof.

2.1.4 The Relationship Between CFSMs and TGs

In this subsection, the following two functions given in the diagram below, will be defined

$$\text{CFSM} \underset{Tc}{\overset{Ct}{\rightleftarrows}} \text{TG}$$

To prove the correctness of the transformation process from SCCS to CFSMs, it is important to prove that the function 'Ct' is bijective. This proof will be given at the end of this subsection.

The function 'Ct' is also important for the derivation of the properties a TG must possess for it to represent a CFSM.

Definition 2.1.5 *Ct:* $\{CFSM\} \rightarrow \{TG\}$

$$\text{Ct}[< Q, X, Y, k, B >] =< Q, X \times Y, k, D >]$$

$D = \{< q,(x,y),q' > |\ q,q' \in Q \wedge (x,y) \in B(q,q')\}$

We now see that for each TG which is an image of a CFSM holds:
there exists a function $p : Q \rightarrow Y$. Such that:
$\{< q,(x,y),q' > \in D \Rightarrow y = p(q)\}$

Definition 2.1.6 *Tc* : $\{TG\} \rightarrow \{CFSM\}$

Given is a TG, with a finite number of states and derivations.
$\mathrm{Tc}[< Q, X \times Y, k, D >] =< Q, X, Y, k, B >$

For each $q,q' \in Q$:
$B(q,q') \equiv \{(x,y) \mid < q,(x,y),q' > \in D\}$

The following will lead to the proof that Ct and Tc are bijective functions. First, it will be proven that Ct is injective and then it will be proven that Tc is injective.

proof: $\mathrm{Tc}(\mathrm{Ct}(< B, X \times Y, k >)) =< B, X \times Y, k >$

$\mathrm{Tc}(\mathrm{Ct}(< Q, X, Y, k, B >))$
$=$ {Definition of 'Ct', B is a $\mid Q \mid$ square matrix}
$\mathrm{Tc}(< Q, X \times Y, k, \{< q,(x,y),q' > |\ q,q' \in Q \wedge (x,y) \in B(q,q')\} >)$
$=$ {Definition of 'Tc'}
$< Q, X, Y, k, B' >$

in which, for each $q,q' \in Q$:
$B'(q,q')$
$=$ {definition}
$\{(x,y) \mid < q,(x,y),q' > \in \{< q,(x,y),q' > |$
$q,q' \in Q \wedge (x,y) \in B(q,q')\} >)\}$
$=$ {equational reasoning}
$B(q,q')$
end proof

proof:

$\mathrm{Ct}(\mathrm{Tc}(< Q, X \times Y, k, D >)) =< Q, X \times Y, k, D >$

$\mathrm{Ct}(\mathrm{Tc}(< Q, X \times Y, k, D >))$
$= \{\text{Def Tc}\}$
$Ct(< \{\forall q, q' \in Q : B(q, q') \equiv \{(x, y) \mid < q, (x, y), q' > \in D\}\}, X \times Y, k >)$
$= \{\text{Def Ct}\}$
$< Q, X \times Y, k, D' >$
where D'
D'
$= \{\text{defintion}\}$
$\{< q, (x, y), q' > \mid q, q' \in Q \wedge (x, y) \in \{(x, y) \mid < q, (x, y), q' > \in D\}\}$
$= \{\text{equational reasoning}\}$
D
end proof

2.2 The Connect Operation

In this section, the Connection Operation will be defined. This section will begin by giving the intuitive meaning of connection. This will be followed by its formal definition and the definition of the operators from which it will be composed. The section will end by proving that the set of all CFSMs (i.e. the set of all implementable and not implemetable CFSMs) is closed under this operation.

The idea behind connecting is the yearn for the ability to decompose a complex behavior, one requiring a large state space to describe, into several, simple behaviors. Connection is, in fact, the inverse of decomposition. After connection, it will be possible to verify the specification of the structure.

2.2.1 The Basic Operators

The definition of the connect operation makes use of the following three operators:

- Φ : Morphism.
- $\lceil$: Projection.

- $\otimes_T$: Concurrent Composition.

This section will begin with their definition and end with a proof that the set of all CFSMs are closed under these three operators.

Definition 2.2.1 A Morphism, Φ, is a function: $\Phi : U_x \times U_y \rightarrow U_x \times U_y$.
which can be expressed as the composition of two functions:
$\Phi_x : U_x \rightarrow U_x$.
$\Phi_y : U_y \rightarrow U_y$.

Hence:
$\Phi(x,y) = (\Phi_x(x), \Phi_y(y))$
Note: $\Phi(\emptyset) = \emptyset$.

Two special types of morphisms will be used in the connection operation, 'Φ_{sh}' and 'Φ_H'.

Definition 2.2.2 Φ_{sh}, 'sh' for 'shuffle', does nothing more than to 'shuffle' subalphabets around. For example:
$\Phi_{sh}(X_1 \times X_2 \times Y) = X_2 \times X_1 \times Y$
'Φ_{sh}' is a morphism since:
$\Phi_{sh}(x,y) = (\Phi_{shx}(x), \Phi_{shy}(y))$

Definition 2.2.3 'Φ_H', 'H' for 'Hide', can be used to 'eliminate' any subalphabet. For example:
$\Phi_H(X_1 \times X_2 \times Y_1 \times Y_2) = X_2 \times Y_1$
'Φ_H' is a morphism since:
$\Phi_H(x,y) = (\Phi_{Hx}(x), \Phi_{Hy}(y))$

Definition 2.2.4 Projection: '$B \upharpoonright X \times Y$'. Where '$B$' is a $\mid I \mid$ square matrix and $X \times Y \subseteq U_x \times U_y$.
$B \upharpoonright X \times Y = \{B(i,j) \bigcap X \times Y \mid i,j$
Hence, if $B : I \times I \rightarrow \mathcal{P}(X_1 \times Y_1)$, then:
$(B \upharpoonright X \times Y) : I \times I \rightarrow \mathcal{P}(X_1 \times Y_1 \bigcap X \times Y)$.

Definition 2.2.5 Concurrent Composition: '$B_1 \otimes_T B_2$', let

$B_1 : I \times I \rightarrow \mathcal{P}(X_1 \times Y_1)$

$B_2 : Q \times Q \rightarrow \mathcal{P}(X_2 \times Y_2)$

$T \subseteq U_x \times U_y$

and there must exist a Φ_{sh} such that:

$T = \Phi_{sh}[X_1 \times X_2 \times Y_1 \times Y_2]$

then:

$(B_1 \otimes_T B_2) \equiv B_3$

where $B_3 : P \times P \rightarrow \mathcal{P}(T)$ and $P \equiv I \times Q$.

The concurrent operator works like the tensor product of matrices. The tensor product is defined operationally as follows:

If A=$[a_{ij}]$is a n by n matrix and B is a m by m matrix, then A$\otimes_T$B is a n·m by n·m matrix.

$$A \otimes_T B = \begin{bmatrix} a_{11} \otimes_T b_{11} & \cdots & a_{11} \otimes_T b_{1m} & \cdots & a_{1n} \otimes_T b_{11} & \cdots & a_{1n} \otimes_T b_{1m} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{11} \otimes_T b_{m1} & \cdots & a_{11} \otimes_T b_{mm} & \cdots & a_{1n} \otimes_T b_{m1} & \cdots & a_{1n} \otimes_T b_{mm} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{n1} \otimes_T b_{11} & \cdots & a_{n1} \otimes_T b_{1m} & \cdots & a_{nn} \otimes_T b_{11} & \cdots & a_{nn} \otimes_T b_{1m} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{n1} \otimes_T b_{m1} & \cdots & a_{n1} \otimes_T b_{mm} & \cdots & a_{nn} \otimes_T b_{m1} & \cdots & a_{nn} \otimes_T b_{mm} \end{bmatrix}$$

And now the problem has been reduced to the tensor product of 1×1 behavior matrices.

$B_1(i,j) \otimes_T B_2(n,m) \subseteq T$

$\emptyset \otimes_T B = \emptyset$

Hence, if $T = X_1 \times X_2 \times Y_1 \times Y_2$, and $(a,b) \in B_1(i,j)$ and $(c,d) \in B_2(n,m)$ then

$(a,b) \otimes_T (c,d) \equiv (a,c,b,d)$

The tensor product can also be expressed in terms of the functions B_1 and B_2 as follows:

For all $i,j \in I$

For all $p,q \in Q$

$$B_3(\mid Q \mid \cdot (i-1)+p, \mid Q \mid \cdot (j-1)+q) = B_1(i,j) \otimes_T B_2(p,q)$$

ENDpq

ENDij

At the end of the section 2.3, it is proven that the concurrent composition operator is commutative.

2.2.2 Closure Proofs

It is very important to prove that the set of all CFSMs, implementable and not implementable, are closed under the three defined operators. The proofs themselves are fortunately simple. The proofs will be given by showing that the operators preserve the three characteristics which define a CFSM. These characteristics are:

(i) A square matrix with a finite number of rows

(ii) The matrix contains a finite number of different actions.

(iii) In each row the output symbol of all 'pairs' in a row are the same.

proof: The morphism operation, Φ.

This operation works only on the actions, hence trait (i) is uneffected.

Trait (ii) is uneffected because it transforms an element in $U_x \times U_y$ to an element in $U_x \times U_y$. This implies that it can never increase the number of unique elements in a matrix. It may, however, increase the size of the set to which the elements belong:

$\Phi[X \times Y] = X_1 \times \ldots \times X_n \times Y$. But, because no configuration partitions its alphabet into an infinite number of subalphabets, (ii) remains intact.

Trait (iii) remains intact because $\Phi[X \times Y] = \Phi_x[X] \times \Phi_y[Y]$. This implies that the new name of the output symbol in a pair can never depend on the new name of its corresponding input symbol.

end proof

proof: The restriction operation, $\restriction$

Trait (i) remains intact because the projection operator only works on symbol pairs.

Projection is an intersection operation, hence the number of different actions will either remain the same or decrease, so that property (ii) and (iii) are not effected.

end proof

proof: The concurrent product.

Let us take as example $B_1 \otimes B_2 = B_3$, where B_1 is a $\mid I \mid$ square matrix and B_2 is a $\mid Q \mid$ square matrix.

The tensor product will result in a $\mid Q \mid \cdot \mid I \mid$ square matrix. Because $\mid Q \mid$ and $\mid I \mid$ are finite, so in their product. Hence, trait (i) remains intact.

The set of symbols in the resulting matrix is the product of the symbols used in B_1 and B_2. Hence, trait (ii) remains intact.

As for trait (iii), an element in B_3 is:

$B_3(\mid Q \mid \cdot (i-1)+p, \mid Q \mid \cdot (j-1)+q) = B_1(i,j) \otimes_T B_2(p,q)$

where $p,q \in Q$ and $i,j \in I$

A row, in B_3, is only a function of the row numbers 'i' and 'p' of B_1 and B_2. It is independent of a parameter corresponding to a column. Because the output symbol in B_1 and B_2 is only dependent on the row, this will then also be the case for an output symbol in B_3.

end proof

2.2.3 The Connection Operation

The connection operation connects a single sub-input alphabet with a single sub-output alphabet. The connections mechanism is very simple, it makes use of a bijective function between symbols of the single input and output sub-alphabet. The connect operation projects the actions in the behavior matrix on a special subset of all actions in the matrix. For each element in this special subset, the components in each element wich correspond to the related sub-alphabets, are related by the bijective function.

All elements in the matrix which are not in this special subset are eliminated. They corresponded to instances of incorrect communication. By eliminating these actions, one effectively eliminates possible transitions of the CFSM which receives input. The eliminated transition represent incorrect responses to a given output symbol. Hence, only the correct ones remain.

Definition 2.2.6 Connect($< Q, X, Y, k, B >, \mathcal{R}(I.O)$)= M.

Where:

- C and M are a CFSM and I and O are sub-alphabets of C. Let $C =< Q, X_1 \times X_2, Y_1 \times Y_2, k, B >$, and let $I = X_2$ and $O = Y_2$. The algorithm will yield the following CFSM:

  M=$< Q, X_1, Y_1, k, N >$,

- $\mathcal{R}(I, O)$ is a 'relation' between an input alphabet and an output alphabet of B. The relation implies that a bijective function exists between symbols in a non-empty subset of the two alphabets.

  Let $Gin \subseteq I$ $\quad Gin \neq \emptyset$
  $Gout \subseteq O$ $\quad Gout \neq \emptyset$

  The relation defines a one-to-one relation between elements in Gin and $Gout$: $f : Gout \rightarrow Gin$, f is bijective. The relation $\mathcal{R}$, is defined as follows:

  $\mathcal{R}(I, O) = \{(x, y) \mid y \in Gout \wedge x = f(y)\}$

The connect operation is defined as:
$N = \Phi_H[\Phi_{sh}(B) \uparrow X_1 \times \mathcal{R}(I, O) \times Y_1]$
where
$\Phi_{sh} : B(J, J) \rightarrow X_1 \times X_2 \times Y_2 \times Y_1$
$\Phi_H : X_1 \times X_2 \times Y_2 \times Y_1 \rightarrow X_1 \times Y_1$

Because the connect operator is a composition of two operators which are closed under all CFSMs, it follows that the connect operator is also closed under all CFSMs.

Take for example, the following behavior matrix. Let us compute the effect of connecting the input data port, D with the inverse output port, $\bar{Q}$.

Connect$(< \{s1, s2\}, D \times T, Q \times \bar{Q}, s1, B >, \mathcal{R}(D, \bar{Q}))$
where $\mathcal{R}(D, \bar{Q}) = \{(0, 0), (1, 1)\}$

Given is the behavior matrix of the D-FF, as defined earlier. After the shuffling we have: $T \times D \times \bar{Q} \times Q$:

	$s1$	$s2$
$s1$	$\{(0,1,1,0),$ $(0,0,1,0),$ $(1,0,1,0)\}$	$\{(1,1,1,0)\}$
$s2$	$\{(1,0,0,1)\}$	$\{(0,1,0,1),$ $(1,1,0,1),$ $(1,0,0,1)\}$

After projection on $T \times \{(1,1),(0,0)\} \times Q$:

	$s1$	$s2$
$s1$	$\{(0,1,1,0)\}$	$\{(1,1,1,0)\}$
$s2$	$\{(1,0,0,1)\}$	$\{(1,0,0,1)\}$

and after the hide operation:

	$s1$	$s2$
$s1$	$\{(0,0)\}$	$\{(1,0)\}$
$s2$	$\{(1,1)\}$	$\{(1,1)\}$

If we wish to connect two CFSM together, say an input sub-alphabet 'F' with a output sub-alphabet of 'E', all we need to do is the following:

Let: F=$< Q, X, Y, k, B >$ and E=$< Q', X', Y', m, C >$,

Connect($< B \otimes_T C, \mid Q \mid \cdot (k-1) + m >, \mathcal{R}(I,O)$)= M.
where I is a sub-alphabet of F and O is a sub-alphabet of E.

The connection mechanism therefore allows one input to be connected to exactly one output and visa-versa.

2.3 Equivalence

The equivalence defined by Milner is based on the notion of bisimulation. Given the relation between TGs and CFSMs, it can be formulated

in terms of CFSMs. The bisimulation of CFSMs $< Q, X, Y, k, B >$ and $< S, X', Y', n, C >$, is a relation R between their rows, which contains their begin state. Let $q, q' \in Q$ and $i, i' \in S$. The relation $R \subseteq Q \times S$, is defined that if

1. $< k, n > \in R$.
2. $< q, i > \in R$.
3. For every $(x, y) \in B(q, q')$ there exists a $(x, y) \in C(i, i')$ for which holds: $< q', i' > \in R$.
4. For every $(x, y) \in C(i, i')$ there exists a $(x, y) \in B(q, q')$ for which holds: $< q', i' > \in R$.

Definition 2.3.1 If $< Q, X, Y, k, B >$ and $< S, X', Y', n, C >$ possess a bisimulation, we express this by stating $< Q, X, Y, k, B > \sim < S, X', Y', n, C >$

As proven by Milner in [2], '$\sim$', is an equivalence relation.

Proposition 2.3.1 *The relation '$\sim$' is a congruence in CFSMs, that is,* $< Q, X, Y, k, B > \sim < S, X', Y', n, C >$ *implies:*

- $< Q, \Phi_x(X), \Phi_y(Y), k, \Phi(B), k > \sim < S, \Phi(X'), \Phi(Y), n, \Phi(C) >$
- $< Q, X, Y, k, B \restriction I \times O > \sim < S, X', Y', n, C \restriction I \times O >$
- $< Q \times J, T_x, T_y, ba, B \otimes_T > \sim < S \times J, T_x, T_y, ca, C \otimes_T >$
  where, $< J, I, O, m, A >$, $T = T_x \times T_y$
  $ba = \mid J \mid \cdot (k - 1) + m$ and $ca = \mid J \mid \cdot (n - 1) + m$

proof

From this proof, it follows that '$\sim$' is a congruence for the connect operation.

It will suffice to proof the only last item, as this one being the most difficult.

It is enough to show that

$\mathcal{R} = \{ < Q \times J, T_x, T_y, ba, B \otimes_T >, < S \times J, T_x, T_y, ca, C \otimes_T > \mid$
$< Q, X, Y, k, B > \sim < S, X', Y', n, C > \wedge$
$< J, I, O, m, A > \in \text{CFSM} \wedge T = T_x \times T_y = X \times I \times O \times Y \}$

is a bisimulation.

Let $B \otimes_T A = Ba$ and $C \otimes_T A = Ca$. Suppose $(x,i,o,y) \in Ba(ba,j)$, then, $(x,y) \in B(k,j')$ and $(i,o) \in A(m,j")$. With the definition of the tensor product:
$(ba,j) = (\mid J \mid \cdot (k-1) + m, \mid J \mid \cdot (j'-1) + j")$.

But since, $< Q,X,Y,k,B > \sim < S,X',Y',n,C >$, it implies that there is a n' such that $(x,y) \in C(n,n')$ and $< Q,X,Y,j',B > \sim < S,X',Y',n',C >$ This implies that:
$(x,i,o,y) \in Ba(ba, \mid J \mid \cdot (n'-1) + j")$ and:
$< Q \times J, T_x, T_y, j, B \otimes_T A > \sim$
$< S \times J, T_x, T_y, \mid J \mid \cdot (n'-1) + j", C \otimes_T A >$

By a symmetric argument, $\mathcal{R}$ is therefore a Bisimulation.

With the definition of Bisimulation, it is now possible to prove a very important trait of the concurrent composition operator, its commutativity.

Given are:
$< Q_1, X_1, Y_1, k_1, B_1 >$ and $< Q_2, X_2, Y_2, k_2, B_2 >$
and $T_x = X_1 \times X_2, T_y = Y_2 \times Y_1$ and $T = T_x \times T_y$

Proposition:
$< P_{12}, T_x, T_y, k_{12}, B_1 \otimes_T B_2 > \sim < P_{21}, T_x, T_y, k_{21}, B_2 \otimes_T B_1 >$

proof:
Let $B_1 \otimes_T B_2 = B_{12}$ and $B_2 \otimes_T B_1 = B_{21}$

A Bisimulation defines a relation between the rows of B_{12} and B_{21}, which contains their begin state. It is possible to show that a bijective function, 'f', exists between their rows such that:
$f : P_{12} \rightarrow P_{21}$, where '$P_{12}$' is a lexographic ordering on $Q_1 \times Q_2$, and P_{21} is that on $Q_2 \times Q_1$.
For all $q, q' \in P_{12}$, holds:
$B_{12}(q,q') = B_{21}(f(q), f(q'))$, where '=' is meant syntactic equivalence.

The function 'f' will be defined by two bijective functions:
$g_{12} : Q_1 \times Q_2 \rightarrow P_{12}$
$g_{21} : Q_1 \times Q_2 \rightarrow P_{21}$

Where:
Let $i_1 \in Q_1$ and $i_2 \in Q_2$, then,
$g_{12}(i_1, i_2) = \mid Q_2 \mid \cdot (i_1 - 1) + i_2$
$g_{21}(i_1, i_2) = \mid Q_1 \mid \cdot (i_2 - 1) + i_1$

And 'f' can now be defined as:
$f(g_{12}(i_1, i_2)) = g_{21}(i_1, i_2)$

Hence, for any $p \in P$, there exists a $i_1 \in Q_1$ and a $i_2 \in Q_2$, such that $p = g_{12}(i_1, i_2)$, so that $f(p) = g_{21}(i_1, i_2)$. Therefore, the bijectiveness of 'f' follows from the bijectiveness of g_{12} and g_{21}.

end proof.

Chapter 3

The Transformation Process

In this chapter, the transformation will be given from an SCCS expression to a structure of Communicating Finite-State Machines. In Chapter 1, SCCS was presented and a normal form was defined. In Chapter 2, the CFSM model was presented with which a structure of finite-state machines can be described. In this Chapter, SCCS and CFSM formalisms will be related through a transformation function; Sc: SCCS→ CFSM.

For a transformation between two formalisms which can express structure, two important criteria must be met. The transformation must be meaning preserving and it must be able to transform structure. The first point implies that the SCCS expression and its CFSM transform must share a similar Transition Graph representation. The second point implies that the function 'Sc' must behave like a *homomorphism*:

Sc(E op_s F)= Sc(E) op_c Sc(F), where E and F are SCCS expressions, 'op_s' is a SCCS operator and 'op_c' is a similar CFSM operator.

The definition of 'Sc' will proceed in two phases. First, the definition of Sc as a meaning preserving transformation will be given. This will be achieved by viewing Sc as a transformer of a structureless SCCS expression to a structureless CFSM. The second phase will involve the extension of Sc to a homomorphism.

3.1 Preserving Meaning

A meaning preserving transformation implies that both a SCCS expression as its TG representation will map to the same CFSM. This can be expressed

in the following commuting diagram:

$$\begin{array}{lcl} \text{SCCS} & \xrightarrow{St} & \text{TG, } \mathcal{A}\text{ct} \\ \text{Sc} \downarrow & & \downarrow \text{Ace} \\ \text{CFSM} & \xleftarrow{Tc} & \text{TG, } U_x \times U_y \end{array}$$

In the commuting diagram, the only function yet to be defined is 'Ace'. 'Ace' is an 'action encoding function'. It is a bijective function which is to relate actions in a SCCS expression to symbol pairs in a CFSM description. It therefore only changes branch labels and does not effect the graph's structure.

Given 'Ace', the function Sc can be defined as:
$\text{Sc}(\{\text{SCCS}\}) = \text{Tc}(\text{Ace}(\text{St}(\{\text{SCCS}\})))$

This section will mainly be dedicated to the definition of the function 'Ace'. Its definition will be constructed in the following two subsections. Once Ace is defined, it will be possible to classify the SCCS expressions which are transformable to a CFSM. This will be achieved in the last subsection, by defining a subclass in SCCS, which is bijectively transformable to CFSMs.

Hence, in the following two subsections, the function 'Ace' will be constructed. The function was difficult to define because of the vast number of possibilities. Thus, in its development, important choices had to be made. Perhaps the most important choice is the desire to express input and output in SCCS. With this possibility, it will then be possible to make clear in SCCS, which actions are to be interpreted as input and which as output. The only task left for 'Ace' is then to translate the input and output actions to respectively input and output symbols in a CFSM.

To introduce the notion of input and output in SCCS, two postulations are needed:

Proposition 3.1.1 *Every action can be uniquely expressed as a product of an input-action and an output-action.*

Proposition 3.1.2 *An input-action may be expressed only by a product of input-actions and an output-action may be expressed only by a product of output-actions.*

These two propositions, together with the fact that ($\mathcal{A}$ct , $\times$, 1) is an Abelian monoid, leads to the postulation that $\mathcal{A}$ct can be divided into two submonoids: ({input-actions}, $\times$, 1) and ({output-actions},$\times$, 1).

The divisibility of an action into an input and output action, reflects the divisibility of a symbol in $U_x \times U_y$ into an input-output symbol pair. The divisibility of an input-action into multiple input actions, reflects the divisibility of an input symbol into a tuple of input symbols. The same analogy holds for an output-action.

The function 'Ace' has therefore the considerable task of differentiating an input-action from an output-action and of recognizing sub-input and sub-output alphabets. To tackle this problem, the development of 'Ace' will proceed in two phases. First, it will be assumed that every label in a Transition Graph, representing an SCCS expression, is only divisible in exactly one input-action and one ouput-action. Given this version of Ace, an natural extension can be made to the case of multiple input and output-actions.

The first phase will be worked-out in the next subsection and the second in the subsection there after.

3.1.1 Encoding Actions

The function 'Ace', under proposition 3.1.1, can be defined as:

Ace: $\mathcal{A}$ct $\rightarrow U_x \times U_y$

In the rest of this subsection, let $Ac \subset \mathcal{A}$ct , be some finite set of actions. According to propostion 3.1.1, the set Ac can be expressed as follows:

$$Ac \subset \{a \times b \mid a \in \{\text{input-actions}\} \wedge b \in \{\text{output-actions}\}\}$$

Furthermore, let Ace(Ac) = $X \times Y$, where $X \times Y \subset U_x \times U_y$

The only problem to be tackled is the recognition of input and output actions and their translation into the cartesian product of an input-output symbol pair.

A possible solution of the recognition problem is to make a syntactical differentiation between an input-action and an output-action. A convention taken from CCS is to suffix and input-action by a '?' and to suffix an output-action by a '!'. Because the set of input and output actions form a monoid, they both include the action '1'. For this reason, this action is not

to be suffixed.

To transform Ac to $X \times Y$, the set of all input and output actions occuring in Ac, must be found and translated to the symbol sets X and Y.

Let:

$In = \{a \mid \exists a \in \{\text{input-actions}\} : \exists b \in \{\text{output-actions}\} : a \times b \in Ac\}$

$Out = \{b \mid \exists b \in \{\text{output-actions}\} : \exists a \in \{\text{input-actions}\} : a \times b \in Ac\}$

Again, this search for input and output actions is very simple because an input-action will be either the symbol '1', or a symbol suffixed by a '?' and an output-action will be either the symbol '1', or a symbol suffixed by a '!'.

With the two encoding functions:

$Ein : In \rightarrow X$

$Eout : Out \rightarrow Y$

Leads to the following defintion of 'Ace':

$$\text{Ace}(Ac) = Ein(In) \times Eout(Out)$$

The definition of the bijective encoding functions, 'Ein' and '$Eout$' will be left to the designer. The symbols used in SCCS are merely to define a relation between agents. The interpretation of these symbols is open. They may be interpreted as numbers or letters; it does not matter.

For example, take the following SCCS expression:

$$\text{fix}_1\{X_1, X_2\}\{ \quad E_i = \sum_{\substack{j \in \{0,1,2\} \\ i \in \{1,2\}}} b_i! \times a_j?; X_1 + b_i! \times a_3?; X_2$$

If the following encoding is chosen,

$a_0? \rightarrow 01 \quad b_1! \rightarrow 1$

$a_1? \rightarrow 10 \quad b_2! \rightarrow 0$

$a_2? \rightarrow 11$

$a_3? \rightarrow 00$

The behavior resembles that of an 'OR' gate. If we only reassign $b_1!$ as '0' and reassign $b_2!$ as '1', the behavior resembles that of an 'AND' gate.

Under two other simple reassignments, the behavior of an 'NAND' and 'NOR' gate may be acquired.

In the next subsection, this translation mechanism will be extended to multiple alphabets.

3.1.2 Multiple Sub-Alphabets

The problem is now the recognition of multiple sub-input and sub-output alphabets. This problem is complex. For example, the user may have in mind that each action in 'Ac' is expressible as a product of four symbols, each from a different alphabet:

$$Ac = \{a_1 \times a_2 \times b_1 \times b_2 \mid a_1 \in In_1, a_2 \in In_2, b_1 \in Out_1, b_2 \in Out_2\}$$

But this implies that, for proper recognition, each symbol in each alphabet must be unique over all alphabets. This implies that we need as many unique symbols as the product of the number of elements in each alphabet. This is a very unhandy method of representing multiple sub-alphabets. A more elegant solution is made possible by the group theory. Each element in the set 'Ac' can be uniquely expressed, up to an order of factors, as a product of *independent* generators:

$$a = \lambda_1^{n1} \times \ldots \times \lambda_k^{nk},\ ni > 0.$$

Two generators, λ_1, λ_2 are independent if $\lambda_1 \times \lambda_2 \neq 1$, and $\lambda_1 \neq \lambda_2$.

Each generator can be associated with a sub-alphabet. There will then be as many generators as sub-alphabets.

Let λ_1^{n1} be equal to either $\lambda_1?n1$ or $\lambda_1!n1$, where $\lambda_1^0 = \lambda_1?0 = 1$. Let the set of all generators occurring in actions in Ac be represented by 'Λ', then:

$Ac \subset \Lambda^*$. We can say that Ac is defined relative to the monoid Λ.

Now each subalphabet can be found as follows:

$$In_i = \{\lambda_i?n \mid \exists rest \in (\Lambda - \lambda_i?)^* : \lambda_i?n \times rest \in Ac\}$$

$$Out_j = \{\lambda_j!n \mid \exists rest \in (\Lambda - \lambda_j!)^* : \lambda_j!n \times rest \in Ac\}$$

The search is again made simple. To find the elements of, for example, In_i, each element in Ac must be searched for occurances of the generator, $\lambda_i?$. If none is found, then the element contained the occurance $\lambda_i?0 = 1$.

Hence, each element in Ac contains, at most, as many unique symbols as generators in Λ. The morphism Ace can now be defined for the given set Ac as follows:

with:

$Ein_i : In_i \rightarrow X_i$

$Eout_j : Out_j \rightarrow Y_j$

$$\text{Ace}(Ac) = Ein_1(In_1) \times \ldots \times Ein_n(In_n) \times Eout_1(Out_1) \times \ldots$$

For example, take the following SCCS expression:

$$\text{fix}_1\{X_1, X_2\}\{ \quad E_i = \sum_{j \in \{1,2\}} b!i \times a?j \times c?1; X_1 + b!i \times a?2 \times c?2; X_2 \quad i \in \{1,2\}\}$$

It describes the behavior of an 'AND' gate under the following encoding:

$a?1 \rightarrow 0 \quad c?1 \rightarrow 0 \quad b!1 \rightarrow 0$

$a?2 \rightarrow 1 \quad c?2 \rightarrow 1 \quad b!2 \rightarrow 1$

We see that the size of the generator set of Ac is very important, since the number of generators is the number of sub-alphabets in the CFSM. Hence the set Ac is not only characterized by the set of actions it contains but also by the minimum set of generators needed to express its actions. A set Ac', of an expression, where $Ac' \subset Ac$ and has less generators than Ac, will be handled differently by the function Ace than it would Ac. Ace will map it to an alphabet with less sub-alphabets.

Given 'Ace', it is now possible to formally define the sought function 'Sc' and to define the subclass of SCCS which is bijectively transformable to a CFSM.

3.1.3 'Sc' as Composition

In this subsection, the function 'Sc' will be formally defined. This will be followed by the definition of the subclass in SCCS which is bijectively

transformable to a CFSM. This subclass will be called FSCCS. The 'F' in the name is to stress that this is a class of finitely representable behaviors.

In the beginning of this Chapter, 'Sc' was defined as:
$Sc(\{SCCS\}) = Tc(Ace(St(\{SCCS\})))$

Now that the three functions, Tc, Ace and St have been defined, three important questions must be ask about the validity of this composition: what is its domain and range, and is it injective, surjective or both.

The question about its range was indirectly answered in Chapter 1. There it was shown that the range of St corresponded with the set of all infinite machines. But because Tc is defined for only finite Transition Graphs, the range of St is 'limited' to the set of all finite state machines.

This limitation of the range, implies that the domain is limited to all SCCS expressions which map to a finite TG. Further restrictions in the domain follow from the characteristics of the functions Tc, Ace and St.

In Chapter 1, it was shown that St is bijective for all Agents. Hence, the only the set of Agents which cannot be represented by a finite TG are those Agents whose proper normal form consists of an infinite number of expressions.

In Chapter 2, it was shown that Tc is bijective for a subset of TG's: namely those, $< Q, X \times Y, k, D >$, for which a function exists: $\alpha : Q \rightarrow Y$ such that,

if $< q, (x, y), q' > \in D$ implies $y = \alpha(q)$.

Furthermore, a Transition Graph was of an implementable CFSM if, for all states, S, which are accessible from the begin state, hold:

$[\forall q \in S : \exists (x, y) \in X \times Y :< q, (x, y), q' > \in D]$

The function Ace, is also bijective, given a subset of $\mathcal{A}ct$ for which it is defined.

From these mentioned characteristics, together with the implementability criterium, follows the definition of a subclass of SCCS, called FSCCS, for which the function Sc is a bijection.

Definition 3.1.1 FSCCS is the set of SCCS expressions which are strongly congruent to a proper normal form exression with the following characteristics:

Given is an expression of the form:
$\text{fix}_k\{X_i \mid i \in I\}\{E_i \mid i \in I\})$, for which the function is defined:
$L : I \times I \rightarrow Ac$,

1. The expression is an Agent.
2. 'I' is a finite set of indices.
3. 'Ac' is a finite set of actions.
4. For every element, $\alpha \in Ac$; $\alpha = \alpha_{in} \times \alpha_{out}$ where α_{in} is an input action and α_{out} is an output action.
5. There must exist a function $p : I \rightarrow \{\text{output-actions}\}$ such that each expression in the $\mid I \mid$ family of expressions, has the form:

$$E_i = \sum_{j \in I} \sum_{\alpha \in L(i,j)} \alpha_{in} \times p(i); X_j$$

6. This proper normal form expression is implementable if for all $j \in I : L(j, I) \neq \emptyset$

We can now give the following formal definition of the function of Sc for an FSCCS expression:

$$Sc(\text{fix}_k\{X_i \mid i \in I\}\{E_i \mid i \in I\}) =< I, X, Y, k, B >$$

Given are: $L : I \times I \rightarrow Ac$, $Ace(Ac) = X \times Y$
These two functions yield:
$B(i,j)$ =Ace$(L(i,j))$,

3.2 The Transformation as Homomorphism

Sc is a homomorphism if SCCS and CFSM share one or more operators which have the 'same effect' on an expression. The following commuting diagram clarifies:

$$\begin{array}{lcl} \text{FSCCS} \times \text{FSCCS} & \xrightarrow{op_s} & \text{FSCCS} \\ \text{Sc}\downarrow & & \downarrow \text{Sc} \\ \text{CFSM} \times \text{CFSM} & \xrightarrow{op_c} & \text{CFSM} \end{array}$$

In this section, it will be shown that for each of the three basic operators in the CFSM model, there exists an operator in FSCCS with the same effect.

In the following three subsections, the FSCCS equivalent of each of the CFSM operators will be given. In the forth subsection, these translations will be joined to form the FSCCS equivalent of the connect operator.

In these subsections, let 'E' and 'F' be FSCCS expressions.

3.2.1 Projection

Let 'A' be the set of actions in expression 'E' and let the function 'Ace' translate these actions to input-output symbol pairs.

$$Sc(E \restriction A) = Sc(E) \restriction \mathrm{Ace}(A)$$

There are no restrictions on its use. The restriction operation clearly has the same effect in both formalisms.

3.2.2 Concurrent Composition

Let the set of actions in expression 'E' be generated by the set 'Λ_e', and the actions in 'F' by the set 'Λ_f' and finally, let the set of actions in $E \times F$ be in Ac.

$$Sc(E \times F) = Sc(E) \otimes_T Sc(F)$$

From the definition of the concurrent operator, we know that the set T is the cartesian product of the alphabets of each CFSM. This implies that the generator set of the actions in the product $E \times F$, must be the sum of the generator sets of each expression individually. Therefore, for a correct transformation the only restriction in FSCCS needed is that the generator sets, 'Λ_e', and 'Λ_f' be disjoint:

$\Lambda_e \cap \Lambda_f = \emptyset$.

Now we can say that: $Ac \subset (\Lambda_e \cup \Lambda_f)^*$

and $\mathrm{Ace}(Ac) = T$

3.2.3 Morphism

$Sc(\Phi^s(E)) = \Phi^c(Sc(E))$

In the CFSM model, a morphism operator must be expressible as the composition of two operators. One which works on the input and the other which works only on the output. An analogous constraint must also hold for the morphism in FSCCS. We can say that the morphism operator, Φ^s is only allowed if it can be expressed as the composition of two morphisms:

Φ^s_x : {input-actions} $\rightarrow$ {input-actions}

Φ^s_y : {output-actions} $\rightarrow$ {output-actions}

Hence:

$\Phi_s(E) = \Phi_x(\Phi_y(E)) = \Phi_y(\Phi_x(E))$

The last important property of a morphism is that is does not rename a generator as the product of an infinite number of generators. Such a morphism would not be translatable to one in the CFSM model.

It must be stressed that a morphism in FSCCS, effects the set of actions, Ac, of an Agent. If it changes the number of generators needed to generate the actions of an Agent, it will thereby change the number of sub-alphabets in the CFSM target machine.

3.2.4 Connect

In the CFSM formalism, the connect operation 'joins' an input and an output alphabet of a single CFSM. Take the CFSM $< Q, X_1 \times X_2, Y_1 \times Y_2, k, B >$ as an example.

$\text{Connect}(< Q, X_1 \times X_2, Y_1 \times Y_2, k, B >, \mathcal{R}(X_2, Y_2))$
$= \Phi^c_H[\Phi^c_{sh}(B) \restriction X_1 \times \mathcal{R}(X_2, Y_2) \times Y_1]$

In FSCCS, this operator has a very natural extension. The connect operator in FSCCS, will take advantage of group theory.

$$\text{Connect}(E, \mathcal{R}(a_2?, b_2!)) = \Phi^s_H[\Phi^s_{sh}(E) \restriction (\Lambda - \{a_2?, b_2!\})^*]$$

Where the set of actions in expression 'E' is 'Ac', and this set is expressible by the generator set: 'Λ', where $a_2?, b_2! \in \Lambda$. For clarity, let $\Lambda = \{a_1?, a_2?, b_1!, b_2!\}$ and let $\text{Ace}(Ac) = X_1 \times X_2 \times Y_1 \times Y_2$

So the connect operation also consists of three operations in FSCCS: first 'shuffle' generators, Φ^s_{sh}, then project and finally 'hide' generators, Φ^s_H. It will be shown that the two morphism operations can be defined such that Sc is a homomorphism for all three operations.

To begin with, the relation $\mathcal{R}(a_2?, b_2!)$ defines a bijective relation between the elements in the following two sets:

$$In_{a2} = \{a_2?n \mid \exists rest \in (\Lambda - a_2?)^* : a_2?n \times rest \in Ac\}$$

$$Out_{b2} = \{b_2!n \mid \exists rest \in (\Lambda - b_2!)^* : b_2!n \times rest \in Ac\}$$

Given are the four encoding functions, Ein_{a1}, Ein_{a2}, $Eout_{b1}$, $Eout_{b2}$ each which translate a set of actions to a set of symbols of a CFSM sub-alphabet. Let:

$Ein_{a1}(In_{a1}) \times Ein_{a2}(In_{a2}) \times Eout_{b1}(Out_{b1}) \times Eout_{b2}(Out_{b2}) = X_1 \times X_2 \times Y_1 \times Y_2$

The 'shuffle' morphism, defined in CFSM, will have a very important function here. Let $c?, c! \notin \Lambda$. The shuffle morphism will rename the elements in In_a to elements in an equally large set: In_c and will rename Out_b to elements in an equally large set: Out_c. These generators will be 'dependent': $c? \times c! = 1$

$$\Phi^s_{sx} : In_{a2} \rightarrow In_c$$
$$\Phi^s_{sy} : Out_{b2} \rightarrow Out_c$$

The shuffle morphism is defined such that:

1. $\Phi^s_{sx}(a?i) \times \Phi^s_{sy}(b!j) = 1$ if and only if
   $(a?i, b!j) \in \mathcal{R}(a?, b!)$.
2. $(c?0, c!0) \in \Phi^s_{sh}(\mathcal{R}(a?, b!))$
3. $Ein_{a2}(In_{a2}) = Ein_c(\Phi^s_{sx}(In_{a2}))$
   $Eout_{b2}(Out_{b2}) = Eout_c(\Phi^s_{sy}(Out_{b2}))$
4. $\text{Ace}(a_1?n \times c!m) = (Ein_{a1}(a_1?n), Ein_c(c?0), Eout_c(c!m), Eout_{b1}(b_1!0))$

Conditions (1) and (2) shows how the group theory is used. The relation $\mathcal{R}(a?, b!)$ is transformed into a relation which confirms the fact that in a group, each element has exactly one inverse element. Condition (2) shows that '1' is also in the group.

Conditions (3) and (4) are needed to insure that Sc is a homomorphism for the morphism operation. Condition (4) shows that

$\text{Ace}(\Phi^s_{sh}(Ac)) = X_1 \times Ein_c(In_c) \times Eout_c(Out_c) \times Y_1$.

With condition (3) we know $X_2 = Ein_c(In_c)$ and $Y_2 = Eout_c(Out_c)$, so that:

$\text{Ace}(\Phi^s_{sh}(Ac)) = X_1 \times X_2 \times Y_2 \times Y_1$

These four conditions insure the correctness of the projection operation:

$Sc[\Phi^s_{sh}(E) \upharpoonright (\Lambda - \{a_2?, b_2!\})^*] = Sc[\Phi^s_{sh}(E)] \upharpoonright Ace[(\Lambda - \{a_2?, b_2!\})^*]$

$\text{Ace}[(a_1?, b_1!)^*] = X_1 \times \{Ein_c(c?0), Eout_c c!0\} \times Y_1$

From condition (2) we know that $(c?0, c!0) \in \Phi^s_{sh}(\mathcal{R}(a?, b!))$ and under the group theory we know that each simultaneous occurrence of a $c?$ and a $c!$, will result in either a '1' or a pair, $c?n \times c!m$, where

$(c?n, c!m) \notin \Phi^s_{sh}(\mathcal{R}(a?, b!))$.

Now for the last operation, the 'Hiding' morphism. In the CFSM model, this morphism 'hides' the two joined alphabets. In our example, if Φ^c_H is the equivalent CFSM hiding morphism, then

$\Phi^c_H(X_1 \times X_2 \times Y_2 \times Y_1) = X_1 \times Y_1$.

We must therefore find an equivalent morphism in FSCCS. To achieve this, we must take advantage of the definition of the function Ace. Now,

$\text{Ace}(\Phi^s_{sh}(Ac)) = X_1 \times X_2 \times Y_2 \times Y_1$

For the morphism Φ^s_H must hold:

$\text{Ace}(\Phi^s_H(\Phi^s_{sh}(Ac))) = X_1 \times Y_1$

The hiding morphism must therefore map the generators $c?, c!$ to unity, thereby eliminating them from the minimum set of generators needed to generate all of the actions in the new expression.

In conclusion, the definition has now been given of the three operations of the connect operator for FSCCS expressions.

For example, take an FSCCS description of a D-FF, given in the connect

example in Chapter 2. Here we will carry out the same connection in FSCCS:

Connect(D-FF, $\mathcal{R}(d?, \bar{q}!)$), where the generators in the expression 'D-FF' are:$\{t?, d?, q!, \bar{q}!\}$, and the specification is the following:

$$\text{D-FF} \equiv \text{fix}_1\{X_i \mid i \in \{0,1\}\}\{$$
$$E_i = q!i \times \bar{q}!inv(i) \times t?1 \times \sum_{j\in\{0,1\}} d?j; X_j$$
$$+q!i \times \bar{q}!inv(i) \times t?0 \times \sum_{j\in\{0,1\}} d?j; X_i$$
$$i \in \{0,1\}\}$$

Notice the use of the function inv, which is a function of the index set $\{0,1\}$. It is defined as: $inv(0) = 1$ and $inv(1) = 0$. It must be stressed that this function is introduced for convenience, otherwise it would be neccessary to 'write-out' the description of both E_0 and E_1.

For the translation to the CFSM description, the following encoding is defined:

$d?0 \to 0 \quad t?0 \to 0 \quad q!0 \to 0 \quad \bar{q}!0 \to 0$
$d?1 \to 1 \quad t?1 \to 1 \quad q!1 \to 1 \quad \bar{q}!1 \to 1$

The relation $\mathcal{R}(d?, \bar{q}!) = \{(d?0, \bar{q}!0), (d?1, \bar{q}!1)\}$. Hence, the shuffle morphism can be defined as:

$\Phi^s_{sx}(d?) = c?$ and $\Phi^s_{sy}(\bar{q}!) = c!$.

Now that the shuffle morphism is defined, we can find Φ^s_{sh}(D-FF):

$$\Phi^s_{sh}(\text{D-FF}) \equiv \text{fix}_1\{X_i \mid i \in \{0,1\}\}\{$$
$$E_i = q!i \times c!inv(i) \times timest?1 \sum_{j\in\{0,1\}} c?j; X_j$$
$$+q!i \times c!inv(i) \times t?0 \times \sum_{j\in\{0,1\}} c?j; X_i$$
$$i \in \{0,1\}\}$$

projected on the set $(d?, q!)^*$

$$\Phi^s_{sh}(\text{D-FF}) \equiv \text{fix}_1\{X_i \mid i \in \{0,1\}\}\{E_i = q!i \times t?1; X_{inv(i)} + q!i \times t?0; X_i$$
$$i \in \{0,1\}\}$$

We see now that the hiding morphism does nothing more than to make explicit that the generator pair, $c!, c?$, is no longer neccessary. For clarity, this expression can be written out. The result here can now be compaired to the result of the connect operation given in Chapter 2.

$$\begin{aligned}
&\mathrm{fix}_1\{X_1, X_0\}\{ \\
&\qquad E_0 = q!0 \times t?1; X_1 + q!0 \times t?0; X_0 \\
&\qquad E_1 = q!1 \times t?1; X_0 + q!0 \times t?0; X_1 \\
&\qquad \}
\end{aligned}$$

Conclusion

In this thesis, importnat advances in SCCS have been made and a formal model of Communicating Finite-State Machines has been developed. Together, these achievements form a definite link between the classical finite-state machine model and a formal, SCCS specification.

The link was constructed in a number of basic steps. One step was to extend the classical finite-state machine model with a 'connect' operation and with a finer equivalence relation. Another step was to construct a framework in SCCS in which essential characteristics of a CFSM can be expressed. This step can be broken down into three smaller ones. The first step was the introduction of a 'normal form'. The identification of a normal form, is interpreted as the identification of a 'structureless' SCCS expression. Given a 'structureless' description as basis, it becomes possible to define what structure is in SCCS. In the next step, the notion of 'input and output' was introduced in a structureless SCCS expression. In the last step, a 'connect' operation was defined in which inputs and outputs can be conneceted, to form a structure.

In conclusion, the link between SCCS and finite-state machines, has resulted in the definition of a CFSM model and the creation of a framework within SCCS, FSCCS. In this framework, it is possible to express the behavior of a structure of finite-state machines. Because SCCS is a synchonous language, it lends itself well for the specification and verification of synchronous system. Hence, important problems in the development of a synchronous circuit, such as timing and performance, can be addressed in FSCCS.

Appendix A

Mathematical Notation

The notation used will be similar to that used in the book of 'EILENBERG'. The notation will be that of functions and monoids.

Given sets X and Y, the notation: f:X→ Y will be used to denote a function defined for all elements in X, x∈X, and with values in Y.

Let the set of all subsets of X, the 'power sets' of X, be denoted by $P(X)$ so that, X∈ $P(X)$ and $\emptyset \in P(X)$ ($\emptyset$ is the 'empty set').

A *relation* f from X to Y, to be written as f:X→Y, is a function:

f:$P(X)\rightarrow P(Y)$.

which is completely additive. This implies that if there is a $f : X \rightarrow Y$, then, if $A \subset X$:

$f(A) = \{f(a) \mid a \in A\}$

Note that f($\emptyset$)=$\emptyset$. Because of the additivity trait, f can be completely defined by its values of single elements in X. Thus, f can be viewed as a function:

f:$X \rightarrow P(Y)$

Definition: A relation, f:$X \rightarrow Y$, is a *partial function* if for each $x \in X$, the set f(x) has at most one element. If it has exactly one element, then f is a *complete* function.

The cartesian product, $X \times Y$, of two sets will denote the set of all pairs (x,y), with $x \in X$ and $y \in Y$. Furthermore, the following sets are isomorphic:

$(X \times Y) \times Z = X \times (Y \times Z) = X \times Y \times Z$

If X is a set, then the number of elements in X will be referred to as:

'$|\ X\ |$'.

An Overview of Group Theory

A group consists of a set G and a function,
*: $G \times G \rightarrow G$
which transforms the pair, (g_1, g_2) to an element, denoted by $g_1 \star g_2$, where $(g_1, g_2) \in G$ and where '$\star$' is some binary operator. The notation of a group will be: (G,$\star$).

The following axioms are postulated:

1. Associativity: for any $g_1, g_2, g_3 \in G$ holds:

   $(g_1 \star g_2) \star g_3 = g_1 \star (g_2 \star g_3)$

2. Identity element: There exists an element $1 \in G$ such that

   $1 \star g = g = g \star 1$

3. Inverse element: For each element $g \in G$ there exists an inverse element $\bar{g} \in G$ such that $g \star \bar{g} = 1$

4. Commutativity: $g_1 \star g_2 = g_2 \star g_1$

A structure (G,*) satisfying:

- axiom 1 is called a semigroup.
- axiom 1 and 2, is called a monoid.
- axiom 1,2 and 3, is called a group.
- axiom 4, is called commutative, or Abelian.

Definition: Let (M,$\star$) be a monoid and let A be a subset in M. $A^{n+1} \equiv A \star A^n$. A^* is defined as:

$$A^* \equiv \bigcup_{n=0}^{\infty} A^n$$

The set $(A^*, \star)$ is a *monoid generated by A*. A^* is freely generated by A if $\forall \alpha \in A^*$, α is *uniquely* expressible as $a_1 \star \ldots \star a_n$, where each a_i is element in A. In this paper, a generator will also be referred to as a 'particle'. This will be made in connection with 'actions' in SCCS.

Appendix B

From SCCS to CCS

In this appendix, the relation between SCCS and CCS will be sketched. Here, only the essential distinctions will be made. For a thorough discussion, see [2].

CCS can be derrived from SCCS by weakening strong bisimulation to weak bisimulation and by allowing only particluate and silent actions.

Allowing only particulate and silent actions, implies that every expression is implicitly projected on $\Lambda \cup \{1\}$, where Λ freely generates $\mathcal{A}ct$.

'Weak Bisimulation' is referred to in CCS, as 'Observation Equivalent'. Two TGs related by strong bisimulation, will always be observation equivalent. They will also remain so if every transition, $q \xrightarrow{a} q'$, is replaced by $q \xrightarrow{a} \Rightarrow q'$, where '$\Rightarrow$' represents zero or more 'silent transitions': '$\xrightarrow{1}$'. In the definition of Weak Bisimulation, the transition symbol '$\stackrel{s}{\Longrightarrow}$' is used. It is defined as:

$$\stackrel{s}{\Longrightarrow} \equiv \begin{cases} \Rightarrow \xrightarrow{s} \Rightarrow & \text{if } s \in \Lambda \\ \Rightarrow & \text{if } s = \epsilon \end{cases}$$

Definition B..1 Weak Bisimulation: Let $C_i =< Q_i, k_i, D_i >$ be a Behavior Graph for i$\in\{1,2\}$. Let $q_1, q_1' \in Q_1$ and $q_2, q_2' \in Q_2$ and $a \in \Lambda$, where Λ freely generates $\mathcal{A}ct$. $R \subseteq Q_1 \times Q_2$ is a bisimulation of C_1 and C_2 if:

1. $< k_1, k_2 > \in R$.
2. $< q_1, q_2 > \in R$.
3. For every $q_1 \stackrel{a}{\Longrightarrow} q_1'$ in D_1 there exists a derivation $q_2 \stackrel{a}{\Longrightarrow} q_2'$ in D_2 for which holds: $< q_1', q_2' > \in R$.

4. For every $q_2 \stackrel{a}{\Longrightarrow} q_2'$ in D_2 there exists a derivation $q_1 \stackrel{a}{\Longrightarrow} q_1'$ in D_1 for which holds: $< q_1', q_2' > \in R$.

If C and D are related by Weak Bisimulation, then C≈D.

The syntax of CCS is:

$$E = a : E \mid \sum_{i \in I} E_i \mid (E \mid E) \mid E \setminus A \mid X \mid \mathrm{fix}_k \tilde{X} \tilde{E}$$

The syntax contains the same constructs, the only difference is that they are given another symbol.

Therefore, CCS is a subset of SCCS in which a weaker equivalence relation is used and whose expression are implicitly projected in the set: $\Lambda \cup \{1\}$.

For the interrested, it is easy to prove that
$P \mid Q \approx P \times Q \restriction (\Lambda \cup \{1\})$.

All what is needed are two propositions:
1)P≈P+1;P
2)P×Q≈ P×Q + P;Q + Q;P

An important question is the effect of the two mentioned changes in SCCS on the 'temporal' meaning of an expression. The answer can be given in a nutshell. In SCCS, time is clearly discrete. Each time 'tick' can be viewed as an integer. To transform to CCS, we must let each 'tick' approach zero. This has as effect, that time, in CCS has a more 'continuous' character. This characteristic is strengthend by not supporting simultaneity in CCS and by the weakening of the equivalence relation. In CCS, the only simultaneous actions allowed are communication actions. A sequence of actions is a partial ordering of events. In SCCS, on the other hand, we not only know their order, but also the time lapse between them.

Appendix C

The Concatenation Operator

To describe the behavior of sequential programs, it is convenient to possess an operator which can express the sequential nature of a behavior. Like the action operator which can be used to sequence actions, $b; c; d; \ldots$, where $b, c, d \in \mathcal{A}\mathrm{ct}$, the concatenation operator can be used to sequence expressions: $A \circ B \circ D \circ \cdots$, where $A, B, D \in \mathcal{E}$. When sequencing behaviors, one must be able to uniquely define when the old expression is to end and when the new one is to begin. This function has been reserved for a single SCCS variable, '$'. This variable will be called 'the concatenation point'.

This operator can be introduced by the following derivation rules:

$$\frac{B \xrightarrow{a} B'}{\$ \circ B \xrightarrow{a} B'} \qquad \frac{B \xrightarrow{a} B'}{B \circ \$ \xrightarrow{a} B'}$$

$$\frac{B \xrightarrow{a} B'}{B \circ C \xrightarrow{a} B' \circ C} \qquad \frac{B_i \circ C \xrightarrow{a} B'}{(\sum_{i \in I} B_i) \circ C \xrightarrow{a} B' \circ C}$$

$$\frac{B \circ D \xrightarrow{a} B' \qquad C \circ D \xrightarrow{a} C'}{(B \times C) \circ D \xrightarrow{a} B' \times C'} \qquad \frac{(B \upharpoonright A) \circ C \xrightarrow{a} B'}{(B \circ C) \upharpoonright A \xrightarrow{a} B'}$$

$$\frac{E_k[\mathrm{fix}\tilde{X}\tilde{E}/\tilde{X}] \circ C \xrightarrow{a} E'}{(\mathrm{fix}_k \tilde{X}\tilde{E}) \circ C \xrightarrow{a} E'}$$

This operator can easily be derived. Let $E, F \in \mathcal{E}$.

Definition: $E \circ F \equiv E[F/\$]$

The concatenation operator binds all *free* occurrences of the variable '$' in the expression E to the expression F. For a complete discussion about *binding* variables, see Chapter 1.

It is important to show that this operator is well defined.

Proposition $(\mathcal{E}/\sim, \circ, \$)$ is a monoid.

proof:

1. $A \circ B$ is an expression if $A, B \in \mathcal{E}$.
2. $(A[B/\$])[C/\$] = A[(B[C/\$])/\$]$ according to the definition of free and bound variables. Thus,

   $(A \circ B) \circ C = A \circ (B \circ C)$
3. $A \circ \$ = A[\$/\$] = A$

   $\$ \circ A = \$[A/\$] = A$

   thus, $\$ \circ A \sim A \sim A \circ \$$

end proof.

Because this operator is a *derived* operator, it is easy to show that the equivalence, $\sim$, is a congruence for this operator. This means:

If $A \sim B$ then $A \circ C \sim B \circ C$ and $C \circ A \sim C \circ B$.

From the given definition, follows these equational properties:

1. $(E + B) \circ C \sim E \circ C + B \circ C$
2. $(E \times B) \circ C \sim (E \circ C) \times (B \circ C)$
3. $(E \restriction A) \circ C \sim (E \circ C) \restriction A$
4. $E_k[\mathrm{fix}\tilde{X}\tilde{E}/\tilde{X}] \circ C \sim (\mathrm{fix}_k\tilde{X}\tilde{E}) \circ C$

Bibliography

[1] Prof.dr.ir. C.J. Koomen. *Systeem Technologie.* Lecture notes, Technische Universiteit Eindhoven, 1985.

[2] Robin Milner. *Calculi for Synchrony and Asynchrony.* University of Edinburgh. 1982.

[3] Robin Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science. 1980.

[4] R.C. Backhouse. *Syntax of Porgramming Languages* theory and practice. Prentice-Hall int. . 1979.

[5] G. Birkhoff and T.C. Bartee. *Modern Applied Algebra.* McGraw-Hill Book Co. . 1970.

[6] S.D. Brookes. *On The Relationship of CCS and CSP.* Automata, Languages and Programming, 10th Colloquium. July 1983. pp. 83-96.

[7] C.A.R. Hoare.*Communicating Sequential Processes.* Prentice-Hall int. .1985.

[8] INMOS Ltd. .*OCCAM Programming Manual.* Prentice-Hall int. .1984.

[9] R. Taylor and D. May. *OCCAM - an overview.* Microprocessors and Microsystems. vol 8 no 1. jan\feb 1984.

[10] A.J Martin. *The Probe: An Addition to Communication Primitives.* Information Processing Letters. nr 20, pp 125-130. 1985

[11] M.A. Arbibi.*Algebraic Theory of Machines, Languages, and Semigroups.* Academic Press. 1968.