1 Introduction

These notes give some further background on those parts of COMP31212 which are not covered in so much depth in the Magee and Kramer course text book [MK06]. They therefore concentrate on the formal semantics of FSP, introduced in Topic 2.3: Some FSP theory and Topic 2.4: More FSP theory.

First an operational semantics for FSP is presented, using a set of transition rules. A semantics for FSP is, of course, given in [MK06] (as an appendix) however it is presented in a different way, by defining a direct translation from FSP term to labelled transition system.

Second, notions of equivalence using bisimilarities are presented, together with an algorithm for calculating a compatible colouring for an FSP process. Again, definitions for strong and weak bisimulation relations are given in an appendix in [MK06] but unfortunately no algorithm for calculating them is documented there.

Important Note: These notes are not stand-alone and only cover a part of the material presented in COMP31212. They augment other material rather than replace it. They should therefore be used in combination with the following resources:

- Magee and Kramer course textbook [MK06], including the suite of applet examples
- the LTSA system
- COMP31212 lecture slides and exercises

2 FSP

The structure of FSP definitions means that only finite-state systems can be defined. This is because any sets must be finite and recursion can only occur within sequential process definitions. Essentially there is only one (top-level) application of parallel composition (combined with action hiding and relabelling operations).

The syntax for FSP includes a number of useful constructs which could be expressed using other more core constructs (so-called ‘syntactic sugar’). This enables more compact FSP
definitions can be constructed, making them easier to understand and maintain\(^1\).

If there is no parallel process definition given in an FSP specification then by default the last sequential process definition listed is used. For example, in the following the default system is taken as the \texttt{COLOUR} process definition.

**Example: Simple FSP Process Definition**

The sequential process definition, \texttt{COLOUR}, first involves a non-deterministic choice involving the \texttt{red} action. One option leads to the constant process \texttt{STOP}, the other to process identifier \texttt{NEWCOLOUR} defined as a local process.

\[
\begin{align*}
\texttt{COLOUR} &= ( \texttt{red} \rightarrow \texttt{STOP} \\
&| \texttt{red} \rightarrow \text{NEWCOLOUR} ), \\
\texttt{NEWCOLOUR} &= ( \texttt{green} \rightarrow \texttt{COLOUR} ).
\end{align*}
\]

3 **Labelled Transition Systems**

A (finite) labelled transition system (LTS) is similar to other descriptions of state-transition systems, such as State Transition Graphs, Finite State Machines and/or Finite Word Automata.

An LTS is represented as directed graph, where each edge (or transition) may take a label. A special state is identified as the start state. Not surprisingly, this is represented as a quadruple:

\[
\text{LTS} = (S, A, \sigma, s_0)
\]

where

- \(S\) is the set of states
- \(A\) the alphabet of labels
- \(\sigma\) transition relation, where \(\sigma \subseteq (S \times A \times S)\)
- \(s_0\) initial state, where \(s_0 \in S\)

For FSP, the set of states \(S\) and alphabet \(A\) are both finite sets.

The transition relation \(\sigma\) is a finite set of triples. Each element describes an edge connecting two states:

\[
(s_f, \text{lbl}, s_t) \in \sigma
\]

where

- \(s_f, s_t \in S\)
- \(\text{lbl} \in A\)

An alternative notation for the transition is ‘\(s_f \xrightarrow{\text{lbl}} s_t\)’, where the arrow/edge is from state \(s_f\), leads to state \(s_t\) and carries the label \(\text{lbl}\).

\(^1\)Strictly speaking, the only ‘core’ FSP construct required to express any LTS is the sequential (unguarded) prefix choice with recursion.
The Labelled Transition System $\text{LTS}$ for the colour process defined above is given by:

$$\text{LTS} = (S, A, \sigma, s_0)$$

where

$S = \{0, 1, 2\}$

$A = \{\text{red, green}\}$

$$\sigma = \{(0, \text{red}, 1), (0, \text{red}, 2), (2, \text{green}, 0)\}$$

$s_0 = 0$

Here, states are represented by integers, and the alphabet, which contains two actions, is represented by strings.

Pictorially the LTS can be presented as:

We could also write the transitions as ‘$0 \xrightarrow{\text{red}} 1$’, ‘$0 \xrightarrow{\text{red}} 2$’, ‘$2 \xrightarrow{\text{green}} 0$’.

### 3.1 The Behaviour of Labelled Transition Systems

A transition path for an LTS gives one possible behaviour of the system. The path starts from the initial state $s_0$ and then consists of a possible sequence of transitions determined by the transition relation for the LTS:

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \ldots$$

In general, a transition path may be infinite.

The complete behaviour of the LTS is then represented by a possibly infinite computation tree. Each transition path is simply one path in the tree.

#### Example: Transition Path and Computation Tree for LTS

For the above LTS example, the following are possible transition paths:

$$\{(0 \xrightarrow{\text{red}} 1), (0 \xrightarrow{\text{red}, \text{green}} 2 \xrightarrow{\text{green}} 0), (0 \xrightarrow{\text{red}, \text{green}} 2 \xrightarrow{\text{green}} 0 \xrightarrow{\text{red}} 1), (0 \xrightarrow{\text{red}, \text{green}} 2 \xrightarrow{\text{green}} 0 \xrightarrow{\text{red}} 2 \xrightarrow{\text{green}} 0 \ldots)\}$$

Part of the (infinite) computation tree is:
4 FSP and LTS

An FSP process term generally consists of FSP operators applied to FSP process constants and identifiers, which are defined by a list of process definitions.

The meaning of an FSP process term, taken in the context of its associated definitions, is defined by translating it to an LTS. The process term therefore represents a state in an LTS. The meaning of each term is determined by defining labelled transitions to new process terms (which of course represent other states in the LTS). The transitions for each FSP construct must therefore be given. It is then possible to define a translation from each FSP term to the LTS which gives its behaviour.

The transition relation $\sigma_P$ for an FSP process term $P$ is:

$$\sigma_P \subseteq (\text{Proc}, \text{Act}, \text{Proc})$$

where $\text{Proc}$ is the set of FSP process terms, and $\text{Act}$ is the set of action labels, together with $\tau$.

An FSP process term $E$ can therefore make a labelled transition into a new FSP process $E_1$, with label $a_1$. A Transition path $E \xrightarrow{a_1} E_1 \xrightarrow{a_2} E_2 \xrightarrow{a_3} \ldots$ for the process term can then be derived.

Example: LTS Transitions

Assuming the previously given COLOUR process definition, a transition path for the process identifier COLOUR is

$$\text{COLOUR} \xrightarrow{\text{red}} \text{NEWCOLOUR} \xrightarrow{\text{green}} \text{COLOUR} \xrightarrow{\text{red}} \text{STOP}$$

Now, instead of naming the states as FSP terms $E, E_1, E_2, \ldots$, we simply map these to some more convenient set of state names, e.g. integers. So COLOUR is 0, NEWCOLOUR is 2 and so on (by convention the starting state is labelled 0 and the 'error' state -1).

5 Transition Rules for FSP Process Terms

The behaviour of each FSP operator is defined via a set of FSP transition rules. These enable the transition relation for any FSP process term to be calculated.
Each transition rule consists of a set of **premisses** and a **conclusion**. If the premisses all hold then the conclusion holds. The conclusion defines a possible transition for an FSP process term. The premisses will either be transitions for sub-terms or will be other Boolean conditions. The transitions for the sub-terms will have been calculated using other transition rules.

The transition rules have the following general format:

\[
\text{RuleName} \quad \text{premiss}_1 \ldots \text{premiss}_n \quad \text{transition}
\]

where \( n \) may be zero, i.e. some transition rules do not require any premisses. This means that when all of the \( n \) premisses are met, then the given transition is a member of the transition relation being determined.

Of course, all transition rules will be applied to FSP process terms which have been defined with respect to a set of process definitions.

### 5.1 Sequential Process Transition Rules

#### 5.1.1 Rule for Prefix

\[
\text{Prefix} \quad (a \rightarrow P) \xrightarrow{a} P
\]

There are **no** premisses required to calculate the transitions for a Prefix term.

For example, the process term ‘(green->COLOUR)’ can make a transition labelled green to the process term COLOUR, i.e. it makes the transition \((green\rightarrow COLOUR) \xrightarrow{\text{green}} COLOUR\), which can be written as the tuple \(((\text{green}\rightarrow COLOUR), \text{green}, COLOUR)\).

Note that Prefix is a special case of **Generalised Guarded Choice**.

#### 5.1.2 Rule for Definition

\[
\text{Defn} \quad P \xrightarrow{a} P' \quad \text{ProcId} = P \quad \text{ProcId}' \xrightarrow{a} P'
\]

For example, given the process definition for process identifier NEWCOLOUR, then this process term can make the transition:

\[
\text{NEWCOLOUR} \xrightarrow{\text{green}} \text{COLOUR}
\]

which could be written as the tuple \((\text{NEWCOLOUR}, \text{green}, \text{COLOUR})\), an element from the transition relation.

This transition can be calculated using the ‘Defn’ transition rule:
\[(\text{green}\rightarrow \text{COLOUR}) \xrightarrow{\text{green}} \text{COLOUR} \quad \text{NEWCOLOUR} \xrightarrow{} \text{COLOUR} \]

since we have already derived the premiss transition for the process term ‘green\rightarrow \text{COLOUR}’ using the Prefix Rule.

### 5.1.3 Rule for Generalised Guarded Choice

\[
\text{for some } i \in \{1, \ldots, n\} : b_i = \text{true}
\]

\[
\text{Choice} \quad \left( \text{when } b_1 \ a_1 \rightarrow P_1 | \cdots | \text{when } b_n \ a_n \rightarrow P_n \right) \xrightarrow{a_i} P_i
\]

The choice can be \textbf{deterministic}, if all of the enabled actions are different, or can be \textbf{non-deterministic} if there is more than one same-name action enabled. Deterministic choice means that just by observing which action is chosen it will be known which continuation process has been reached. On the other hand, non-deterministic choice means that an observer will not know which continuation process had been reached, i.e. which component of the choice has been taken.

For example, the unguarded, non-deterministic, choice

\[
\left( \text{red} \rightarrow \text{STOP} \qquad | \qquad \text{red} \rightarrow \text{NEWCOLOUR} \right)
\]

can be written as the generalised guarded choice given below.

\[
\left( \text{when (1) red} \rightarrow \text{STOP} \qquad | \qquad \text{when (1) red} \rightarrow \text{NEWCOLOUR} \right)
\]

The transition

\[
\left( \text{when (1) red} \rightarrow \text{STOP} \quad | \quad \text{when (1) red} \rightarrow \text{NEWCOLOUR} \right) \xrightarrow{\text{red}} \text{STOP}
\]

can be derived using the generalised guarded choice transition rule, choosing the first element of the guarded choice (both are enabled).

The Boolean guards are logical expressions or integers, where zero is false and non-zero is true (as in the C programming language) — see the appendix.

The Prefix Rule is a special case of Generalised Guarded Choice, where \( n = 1 \) and \( b_1 = \text{true} \).

### 5.1.4 Rule for Hiding

Hiding, or the dual interface operator, restricts which actions are visible outside the process term, replacing any actions to be hidden by the special ‘tau’ action. Hiding describes the actions to be hidden, while the interface operator describes the actions to be left visible:
Example: Hiding

Consider the following process definition:

\[
P = ( \begin{array}{l} a \rightarrow P \\ b \rightarrow P \end{array})
\]

\[
Q = ( \begin{array}{l} a \rightarrow b \rightarrow Q \end{array})
\]

\[
R = ( \begin{array}{l} b \rightarrow a \rightarrow R \end{array})
\]

\[
||SYS1 = (P || Q) \setminus \{a\}
\]

\[
||SYS2 = (SYS1 || R)
\]

\((P || Q)\) has the transition sequence \((P || Q) \xrightarrow{a} (P || (b \rightarrow Q)) \xrightarrow{b} (P || Q)\) and \(a \in \{a\}\), therefore the Hiding rules give:

\[
\text{Hide2} \quad \frac{P || (b \rightarrow Q) \xrightarrow{a} (P || (b \rightarrow Q))}{(P || Q) \setminus \{a\} \xrightarrow{\tau} (P || (b \rightarrow Q)) \setminus \{a\}}
\]

followed by

\[
\text{Hide1} \quad \frac{(P || (b \rightarrow Q)) \setminus \{a\} \xrightarrow{b} (P || Q)}{(P || (b \rightarrow Q)) \setminus \{a\} \xrightarrow{b} (P || Q) \setminus \{a\}}
\]

Because action \(a\) is no longer visible, then process \(R\) does not need to synchronise with it in \(SYS2\), which has the following LTS:

![LTS Diagram](image)

5.2 Process Alphabets

Before moving on to the semantics for composite processes, the alphabet of a process \(P\), \(\alpha(P) \subseteq Label\) must be defined.
This is because the definition for parallel composition requires all element processes to **synchronise** on any actions which they can each **potentially** perform, i.e. all those capable of a transition with the shared action must be able to make that transition at the same time. Otherwise **none** of the separate process will be able to execute that transition.

The alphabet represents the transitions in which a process can engage and is defined recursively on its structure:

- **Guarded Prefix I:** \( \alpha(\text{when(true)}a\rightarrow P) = \{a\} \cup \alpha(P) \)
- **Guarded Prefix II:** \( \alpha(\text{when(false)}a\rightarrow P) = \{\} \)
- **Choice:** \( \alpha(P_1|\ldots|P_n) = \alpha(P_1) \cup \ldots \cup \alpha(P_n) \)
- **Definition I:** \( \alpha(\text{ProcId}) = \alpha(P) \) where \( \text{ProcId} = P \) (unless \( \text{ProcId} \) has already been seen)
- **Definition II:** \( \alpha(\text{ProcId}) = \{\} \) (otherwise)
- **Extension:** \( \alpha(P + \{a_1, \ldots, a_n\}) = \alpha(P) \cup \{a_1, \ldots, a_n\} \)
- **Hiding:** \( \alpha(P\setminus A) = \alpha(P) - A \)
- **Relabel:** \( \alpha(P/\{b/a\}) = \alpha(P)/\{b/a\} \) (where \( P \) is sequential)
- **Parallel:** \( \alpha((P_1|\ldots|P_n)/\{b/a\}) = \alpha(P_1/\{b/a\}) \cup \ldots \cup \alpha(P_n/\{b/a\}) \)
- **Stop:** \( \alpha(\text{STOP}) = \{\} \)

Note that:

- in a guarded prefix, if the guard is false then neither the prefix action nor the actions in the continuation process are included in the alphabet,

- in a parallel composition, the effect of any relabelling must be taken into account before calculating the alphabet.

- in a recursive process definition, the calculation of the alphabet will stop when a call to an already-seen process identifier is encountered

**Example: Calculation of Process Alphabet**

Consider the alphabet of \( SYS \) given the following process definitions:

\[
P = ( a \rightarrow P \\
| \quad \text{when (0) b } \rightarrow P \\
| \quad \text{when (1) c } \rightarrow P \\
).
\]

\[
Q = ( b \rightarrow Q \\
| \quad d \rightarrow Q \\
).
\]

\[
||SYS = ( P || Q ) / \{d/a\}.
\]
\[ \alpha(SYS) = \alpha((P||Q)/\{d/a\}) \]
\[ \alpha((P||Q)/\{d/a\}) = \alpha(P)/\{d/a\} \cup \alpha(Q/d/a) \]
\[ \alpha(P) = \alpha(a\rightarrow P \mid \text{when}(0)b\rightarrow P \mid \text{when}(1)c\rightarrow P) \]
\[ = \alpha(a\rightarrow P) \cup \alpha(\text{when}(0)b\rightarrow P) \cup \alpha(\text{when}(1)c\rightarrow P) \]
\[ \alpha(a\rightarrow P) = \{a\} \cup \alpha(P) \]
\[ = \{a\} \cup \{\} \]
\[ \alpha(\text{when}(0)b\rightarrow P) = \{\} \]
\[ \alpha(\text{when}(1)c\rightarrow P) = \{c\} \]
\[ \text{so } \alpha(P/\{d/a\}) = \{d, c\} \]
\[ \alpha(Q/\{d/a\}) = \alpha(Q)/\{d/a\} \]
\[ \alpha(Q) = \alpha((b\rightarrow Q \mid d\rightarrow Q)) \]
\[ = \alpha(b\rightarrow Q) \cup \alpha(d\rightarrow Q) \]
\[ = \{b, d\} \]

so \( \alpha(SYS) = \{b, c, d\} \)

### 5.2.1 Rule for Parallel Composition

It is now possible to define the transition rules for parallel composition:

\[ \begin{align*}
\text{Par1} & : P \xrightarrow{a} P' \quad a \notin \alpha(Q) \\
\text{Par2} & : Q \xrightarrow{b} Q' \quad b \notin \alpha(P) \\
\text{Par3} & : P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q' \\
\end{align*} \]

### 5.2.2 Rule for Process Labelling

Process labelling is part of the syntactic ‘sugar’ of FSP, allowing more compact process descriptions to be given. Transition rules can be given which essentially define the syntactic translation required to expand to the full process term being described:

\[ \begin{align*}
\text{ProcLabel1} & : a \in A \text{ and } P \xrightarrow{b} P' \\
A :: P \xrightarrow{a\cdot b} A :: P' \\
\text{ProcLabel2} & : P \xrightarrow{b} P' \\
a : P \xrightarrow{a\cdot b} a : P' \\
\text{ProcLabel3} & : a_1 : P | \cdots | a_n : P \xrightarrow{b} P' \\
\{a_1, \ldots, a_n\} : P \xrightarrow{b} P' \\
\end{align*} \]
5.2.3 Rule for Relabelling

\begin{align*}
\text{ReLabel1} & : (P \xrightarrow{b_i} P') \text{ and } map = \{a_1/b_1, \ldots, a_n/b_n\} \\
& \quad \Rightarrow (P/map) \xrightarrow{a_i} (P'/map) \\
\text{ReLabel2} & : (P/map)(Q/map) \xrightarrow{a} (P'||Q') \\
& \quad \Rightarrow (P||Q)/map \xrightarrow{a} (P'||Q')
\end{align*}

5.2.4 Rule for Priority

For actions in a set $B$ with High Priority, at any particular state a process will perform a High Priority action if at all possible, excluding any transitions via non-prioritised actions. Only if there are no High Priority actions available will it perform a transition with a non-prioritised label:

\begin{align*}
\text{Priority I} & : P \xrightarrow{a} P' \text{ and } (a \in B \text{ or } (\forall b \in B \cdot P \xrightarrow{b} )) \\
& \Rightarrow P<<B \xrightarrow{a} P'<<B
\end{align*}

(note: $P \xrightarrow{b}$ is true if process term $P$ cannot make a transition labelled with $b$).

The converse is true for Low Priority actions. They will only get performed if there is no other non-priority action to perform:

\begin{align*}
\text{Priority II} & : P \xrightarrow{a} P' \text{ and } (a \notin B \text{ or } (\forall b \notin B \cdot P \xrightarrow{b} )) \\
& \Rightarrow P>>B \xrightarrow{a} P'>>B
\end{align*}

5.3 Calculating the Behaviour of FSP Process Terms

The behaviour of an FSP process term is unambiguously defined via the transition rules, which are used to calculate the transition relation for the LTS which represents the process. However, applying the transition rules manually and showing all of the working is time-consuming and quickly leads to large derivation ‘trees’ even for relatively small process terms (see below).

It is sometimes desirable to proceed slightly less formally, simply by omitting some of the working in transition calculation, and relying our understanding of how a process term will behave (which of course is obtained by experience of applying the transition rules). In this case, it is useful to recognise that the basic general structure of an FSP process term is

- a top-level parallel composition of calls to previously defined (sequential) process definitions, possibly with the application of hiding and relabelling operators. However, the structure of the parallel process term will remain static, with each component process changing ‘state’.
• a sequential process definition consisting of a choice between actions (or transitions), each leading to a new process (or state). It may be useful to explicitly label each continuation process term, by creating a new process definition to represent it.

If the states of each sequential process can be labelled in a consistent way, then the state of the overall composite system can be represented by a vector of these labels.

Example: Informal Transition Calculation

Consider the following process:

\[ P = ( a \rightarrow b \rightarrow P \mid c \rightarrow a \rightarrow P ). \]
\[ Q = ( d \rightarrow a \rightarrow Q \mid e \rightarrow Q ). \]
\[ ||PARSYS = ( P || Q ) / \{ c/e \}. \]

For convenience, we could rewrite this to the following structurally equivalent process, where each state can be labelled by a vector of process identifiers. The relabelling has also been removed:

\[ P = ( a \rightarrow P_1 \mid c \rightarrow P_2 ), \]
\[ P_1 = ( b \rightarrow P ), \]
\[ P_2 = ( a \rightarrow P ). \]
\[ Q = ( d \rightarrow Q_1 \mid c \rightarrow Q ), \]
\[ Q_1 = ( a \rightarrow Q ). \]
\[ ||PARSYS = ( P || Q ). \]

In other words, we have named each state with a process id.

The transitions can now be calculated, and the states represented as:

\[ (P, Q), (P, Q_1), (P_1, Q), (P_1, Q_1) \]
\[ (P_2, Q), (P_2, Q_1) \]

It is also useful to calculate the alphabets of the component processes, in order to identify the sets of actions which must be synchronised. In the above example, the processes only need to synchronise on action a.

Example: Applying the Transition Rules

The 3 transitions for the \textit{COLOUR} process can be calculated as follows. The starting process can non-deterministically choose between two transitions:
where $COLOUR = (red\rightarrow STOP \mid red\rightarrow NEWCOLOUR)$.

There are no transition rules defined for $STOP$ and so it is not possible for it to make any transition.

Having made a transition to the $NEWCOLOUR$ process, there is now only one action possible:

In the previous LTS for $COLOUR$, state 0 represents $COLOUR$, state 1 represents $STOP$ and state 2 represents $NEWCOLOUR$.

**Example: Applying the Transition Rules Again**

The LTS for process $PARSYS$ has 8 transitions and 6 states. The following gives the full calculation for two of the transitions:

where:

$Q = (d\rightarrow Q1 \mid c\rightarrow Q)$

$PARSYS = (P||Q)$
where

\[ P = (a \rightarrow P_1 \mid c \rightarrow P_2) \]
\[ Q_1 = (a \rightarrow Q) \]

The correct calculation of an LTS can easily be checked by using the LTSA.

## 6 Generating Labelled Transition Systems from FSP

Given the definition, via the above transition rules, of the meaning of each FSP construct, it is now straightforward to define the LTS representing the behaviour of an FSP process. To find the LTS for FSP process \( P \):

1. calculate transition relation \( \sigma \) for \( P \) using the transition rules:

   \[
   \sigma = \{(s_1, a_1, s'_1), \ldots, (s_n, a_n, s'_n)\}
   \]

2. States: \( S = \bigcup_{i=1}^{n} \{s_i\} \cup \{s'_i\} \)

3. Starting state: \( s_0 = P \)

4. Alphabet: \( A = \alpha(P) \)

In practice, the set of states \( S \) is given as a finite range of the integers, rather than as a collection of process terms. Each integer then maps to a particular process term.

## 7 Equivalence of FSP Processes

Two FSP process terms are deemed to be equivalent when they cannot be distinguished by composing with any third process term. In other words their ‘visible’ behaviours are indistinguishable. This definition is chosen so that processes which have ‘trivial’ differences cannot be distinguished, but those that have differences which might affect behaviour when composed with other process term will be distinguishable.

Since the behaviour of an FSP process is represented by an LTS then equivalences between FSP processes are defined via LTSs.
7.1 Strong Bisimilarity

Processes P and Q are equivalent, i.e. \((P =_{\text{equiv}} Q)\) if their LTSs are strongly bisimilar:

\[(\text{lts}(P) \sim \text{lts}(Q))\]

LTSs are then (strongly) bisimilar if they have

1. a compatible colouring,
2. the same alphabets (why is this necessary?)

First, a colouring of an LTS is simply an assignment of a ‘colour’ to each node\(^2\).

Now, the colourings of one or more coloured LTSs are compatible if:

1. The starting states of distinct LTSs have the same colour.
2. Whenever states \(P, Q\) (of the same or distinct LTS) have the same colour then for any transition \(\alpha \rightarrow P'\) there is a transition \(Q \rightarrow Q'\) where \(P', Q'\) have the same colour.

7.2 Computing Colourings

The following is known as the Kanellakis-Smolka algorithm for calculating a partitioning of a set of states in an LTS.

```plaintext
// Calculates partition of set of states States with transitions Transitions
P := \{States\};
changed := true;

while changed do
    changed := false;
    for each B in P do
        for each \(\alpha\) in actions do
            if \(\text{split}(B, \alpha, P) \neq \{B\}\) then
                P := (P\{B\}) \cup \text{split}(B, \alpha, P);
                changed := true;
            end if;
        end for
    end for

        where \(\text{split}(B, \alpha, P) = \{B\}\) if there is no \(\alpha\)-splitter \(B'\) in \(P\)
        \(\{B_1, B_2\}\) otherwise
        \(B_1 = \{b | b \in B \wedge b' \in B' \wedge (b, \alpha, b') \in \text{Transitions}\}\)
        \(B_2 = \{b | b \in B \wedge b' \in B' \wedge (b, \alpha, b') \notin \text{Transitions}\}\)
```

\(^2\)A ‘colour’ assignment does not have to literally involve colours! It simply requires each node to be assigned one of a set of labels. For example, sets of letters or integers can be used. The colouring describes the partition of the set of states of the LTS, i.e. all same-colour states are within one element of the partition and different-colour states are in different elements of the partition.
(note: for two sets \( R, S \) then \( R \setminus S \) gives the set difference — all the elements from \( R \) that are not in \( S \))

The calculation of the split function produces a singleton set precisely when one of the sets \( B_1 \) and \( B_2 \) would be empty (i.e. when \( B \) is completely partitioned with respect to \( \alpha \)).

The algorithm can be described informally as follows. The partition is described by labelling each subset in the partition with a ‘colour’

1. Start LTS with all states labelled the same colour, i.e. there is a trivial partition with just one set.
2. Pick:
   - a set of starting states with a particular colour: \( S_C \)
   - an action \( a \)
   - a colour \( C_{end} \) for the end states

   such that some of the starting states in \( S_C \) can make a transition with the chosen action \( a \) to an end state with colour \( C_{end} \), and the rest of the starting states cannot.
3. Give those states from \( S_C \), which can make a transition with action \( a \) to one of the states colour \( C_{end} \), a new colour
4. Repeat from step two until no further colour/action combinations meet the criteria listed there.

Example: Colouring Example

Consider the following FSP definition:

\[
P = ( \quad a \rightarrow P1 \\
| \quad a \rightarrow P2 \\
| \quad d \rightarrow P ), \\
P1 = ( \quad b \rightarrow d \rightarrow P \\
| \quad c \rightarrow P3 ), \\
P2 = ( \quad b \rightarrow P3 \\
| \quad c \rightarrow P4 ), \\
P3 = ( \quad d \rightarrow P5 ), \\
P4 = ( \quad d \rightarrow P5 ), \\
P5 = ( \quad d \rightarrow P5 \\
| \quad a \rightarrow P1 ).
\]

This has the following LTS:
The colouring for this LTS can now be calculated:

Using ‘colours’ A, B, C, starting with all states 0–6 coloured A:

<table>
<thead>
<tr>
<th>Start Colour/Set</th>
<th>Action</th>
<th>Destination State</th>
<th>Partition of Start Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In fact there are no splits of the current partitions possible, giving the final colouring as:

\[ A : \{3, 4, 6\} \]
\[ B : \{0, 5\} \]
\[ C : \{1, 2\} \]

**Example: A Second Colouring Example**

The FSP process SYS2 is defined as follows:

\[
P = ( a \rightarrow b \rightarrow P \\
    | c \rightarrow Q[0] ), \\
Q[i:0..2] = ( \text{when } (i < 2) d[i] \rightarrow ( c[i] \rightarrow Q[i+1] \\
    | \text{red } \rightarrow b \rightarrow Q1 ) \\
    | d \rightarrow P ), \\
Q1 = ( \text{blue } \rightarrow \text{yellow } \rightarrow \text{red } \rightarrow Q1 ).
\]

\[
R = ( e \rightarrow f \rightarrow g \rightarrow R ), \\
S = ( g \rightarrow e \rightarrow S ).
\]
\[ \text{SYS1} = P / \{ \text{red/a, f/b, green/d}[i:0..2], \text{red/c}[i:0..2]\}. \]
\[ \text{SYS2} = ( \text{SYS1} || R || S ) / \{ \text{red/c, green/d} \}. \]

It is equivalent to the \text{COLOUR} process defined earlier and therefore its LTS can be coloured using 3 colours to represent the 3 sets of bisimilar states.

**Example: A Third Colouring Example**

Consider the following FSP process definition:

\[
P = ( a \rightarrow Q
\mid a \rightarrow R ),
Q = ( b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow P ),
R = ( b \rightarrow c \rightarrow d \rightarrow e \rightarrow g \rightarrow P ),
\]

Each node in its LTS has a different colour. It is not possible to give the same colour to any of the pairs of states performing \( b, c, d \) or \( e \) actions because of the different actions \( f, g \) which return to the starting state.

### 7.3 Minimising LTS

An LTS may have a non-trivial self-compatible colouring, i.e. there are distinct states that get the same colouring. The LTS can be simplified to a smaller (bisimilar) LTS, by identifying states that have the same colour. The LTS is \textit{minimised} when every self-compatible colouring is trivial:

- LTS \( T \) not minimised? Then it has a non-trivial colouring.
- Identify states with the same colour to form \( T_1 \). Then \( T \sim T_1 \).
- If \( T_1 \) is minimised then finish.
- If not we can repeat the procedure to get \( T_2 \) with \( T \sim T_1 \sim T_2 \).
- Termination: \( T_i \) has strictly fewer states than the previous LTS \( T_{i-1} \)

**Example: Minimisation**

The following LTS is the minimised version of the one given above after states in each of the partitions (0,5), (1,2) (3,4,6) have been merged:
7.4 Weak Bisimilarity

A strong bisimulation considers all actions as ‘visible’. However, this means that ‘tau’ actions, which result from hiding, are treated just the same as any other action, even though they represent ‘invisible’ or ‘internal’ behaviour. It therefore may distinguish between processes which differ only because they have different sequences of tau actions.

For example consider the following two process definitions:

\[ P = ( a \rightarrow b \rightarrow \text{STOP} ). \]
\[ Q = ( a \rightarrow Q1 ), \]
\[ Q1 = ( c \rightarrow Q1 \mid b \rightarrow \text{STOP} ) \setminus \{ c \}. \]

Hiding of action c means that Q can make a transition under a, then perform an arbitrary number of tau actions before performing a b action. Its observable behaviour can therefore be considered to be the same as that of process P.

The definition of Weak Bisimilarity is similar to the one for strong bisimilarity, except that it ignores sequences of tau actions.

Notice however that weak bisimulation is no longer a congruence — equivalence is not preserved by the choice operator.

Refer to the Lecture slides for the definition of weak bisimulation.

8 Background

Strong bisimulation, the silent action τ, and weak bisimulation were first introduced by Robin Milner in his process algebra Calculus of Communicating Systems, CCS (see [Mil89]). He also presents the semantics of CCS operationally using transition rules, and gives a formal translation from CCS process terms to ‘flow graphs’ (very similar to LTS).

The main differences between CCS and FSP are

- CCS can define infinite-state systems
• actions in CCS are defined in pairs
• synchronisation between actions in different elements from a parallel composition is point-to-point. Parallel composition alone does not enforce synchronisation between in different process components. This is achieved by using the hiding operator.

CSP is one of the main competing approaches, which was developed by Tony Hoare during the same period[Hoare 85]. Unlike CCS and FSP, CSP uses a ‘failures-divergence’ semantics, which labels each node with the sets of actions on which it fails. The FSP parallel composition operator is similar to one of those used in CSP. Again, CSP may be infinite-state.

9 Acknowledgement

Donal Fellows has prepared the notes on the Colouring algorithm, and produced several of the FSP equivalence examples.

He has also prepared the Java classes for representing an LTS, to form the basis of an implementation of the colouring algorithm.

References


A Summary of Syntax

| Action labels | $a, a_1, a_2, \ldots \in ALabel$ |
| Process       | $P, P_1, P_2, \ldots \in \text{Proc}$ |
| Process Id     | $id, id_1, \ldots$ |
| Boolean expressions | $b, b_1, b_2, \ldots$ |
| Expressions    | $e, e_1, e_2, \ldots$ |
| Variables      | $x, x_1, x_2, \ldots$ |
| $ALabel = Action label$ | $id\ldots id_2$ |
|                | $id[range]$ |
|                | $[range]$ |

$actset, as \in \text{Action set}$

$\{a_{\text{lbs}1}, \ldots, a_{\text{lbs}n}\}$

$[\text{range}]$

$a_1, \ldots, a_n$

$a_{\text{lbs}}\cdot a$

$a_{\text{lbs}}\cdot \text{actset}$

$a_{\text{lbs}}\cdot [\text{range}]$

$e_1, \ldots, e_n$

$x : [e_1, e_n]$

$x : \text{actset}$

University of Manchester 19 Version 1.42, March 2011
### Forms of process expression

<table>
<thead>
<tr>
<th>Category</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stop, error processes</td>
<td>STOP, ERROR</td>
</tr>
<tr>
<td>prefix action</td>
<td>a→P</td>
</tr>
<tr>
<td>examples</td>
<td>(coffee → DRINKS)</td>
</tr>
<tr>
<td>guarded action</td>
<td>when b a→P</td>
</tr>
<tr>
<td></td>
<td>(when (i==0) beep → STOP)</td>
</tr>
<tr>
<td>deterministic choice</td>
<td>a₁→P₁</td>
</tr>
<tr>
<td></td>
<td>(red → COFFEE</td>
</tr>
<tr>
<td>non-deterministic choice</td>
<td>a→P₁</td>
</tr>
<tr>
<td></td>
<td>(toss → HEADS</td>
</tr>
<tr>
<td>dependent process</td>
<td>a[x]</td>
</tr>
<tr>
<td></td>
<td>(out[i] → BUFF)</td>
</tr>
<tr>
<td>indexed choice</td>
<td>a[range]</td>
</tr>
<tr>
<td></td>
<td>(in[i:0..2] → BUFF[i])</td>
</tr>
<tr>
<td>process name</td>
<td>Id</td>
</tr>
<tr>
<td>indexed process name</td>
<td>Id[e₁,...,eₙ]</td>
</tr>
<tr>
<td>parameterised process name</td>
<td>Id(e₁,...,eₙ)</td>
</tr>
</tbody>
</table>
|                                  | DRINKS
|                                  | BUFF[i]
|                                  | CELL(Y) |
| Composite process                 | (P₁||...||Pₙ)          |
|                                  | (CELL(1) || CELL(2) || BUFF) |
| Process labelling                 | actset : P            |
|                                  | a:P                   |
|                                  | { a,b }::P             |
| Alphabet extension                | P + actset            |
| Action relabelling                | P/ {as₁/as₂,...,asₙ/asₙ₊₁} |
|                                  | P / { a/b, c/d }       |
| Action hiding                     | P\actset              |
|                                  | P a,b                 |
|                                  | P @ { c,d }           |
| Process equation                  | pnm = P               |
|                                  | TICK = ( tick → TICK) |
| Process definition                | declarations          |
|                                  | main_process_equation, |
|                                  | local_process_equation₁, |
|                                  | :                     |
|                                  | local_process_equationₙ, |
|                                  | |SYS = (P || Q) |
| Composite definition              | ||Id = composite_process |
| Conditional                       | if e then P else P    |
|                                  | if (i==15) then DRINKS|
|                                  | else CHANGE[i]        |
| Replication                       | forall indexrange compbody |
|                                  | forall [i:0..3] a[i]:BUFF |
| Priority                          | P >> actset           |
|                                  | P >> a,b              |
|                                  | P << c,d              |

### Operational Semantics: Transition Rules

**Prefix**

\[(a→P) \xrightarrow{a} P\]

**Defn**

\[P \xrightarrow{\text{ProcId}} P'\]

where (ProcId = P)

**Choice**

for some \(i \in \{1,...,n\}\)

\[(a₁→P₁ | · · · | aₙ→Pₙ) \xrightarrow{a} P_i\]

**Par1**

\[P \xrightarrow{a} P' \text{ and } a \notin \alpha(Q)\]

\[(P||Q) \xrightarrow{a} (P'||Q)\]

**Par2**

\[Q \xrightarrow{b} Q' \text{ and } b \notin \alpha(P)\]

\[(P||Q) \xrightarrow{b} (P||Q')\]
Par3. \( P \leadsto_a P' \) and \( Q \leadsto_a Q' \)

\( (P||Q) \leadsto_a (P'||Q') \)

ProcLabel1. \( a : P \overset{b}{\rightarrow} P' \)

\( a : P \overset{a : b}{\rightarrow} a : P' \)

ProcLabel2. \( a \in A \) and \( P \overset{b}{\rightarrow} P' \)

\( A : P \overset{a : b}{\rightarrow} A : P' \)

ReLabel1. \( (P \overset{b_i}{\rightarrow} P') \) and \( \text{map} = \{a_1/b_1, \ldots, a_n/b_n\} \)

\( (P/\text{map}) \overset{a_i}{\rightarrow} (P'/\text{map}) \)

ReLabel2. \( (P/\text{map}||Q/\text{map}) \overset{a_i}{\rightarrow} (P'||Q'/\text{map}) \)

Hide1. \( P \overset{b}{\rightarrow} P' \) and \( b \not\in A \)

\( P \overset{b}{\rightarrow} P' \)

Hide2. \( P \overset{b}{\rightarrow} P' \) and \( b \in A \)

\( P \overset{b}{\rightarrow} P' \)

Hide3. \( P \overset{b}{\rightarrow} P' \) and \( b \in A \)

\( P @ A \overset{b}{\rightarrow} P' @ A \)

Hide4. \( P \overset{b}{\rightarrow} P' \) and \( b \not\in A \)

\( P @ A \overset{\tau}{\rightarrow} P' @ A \)

Priority I. \( P \overset{a}{\rightarrow} P' \) and \( (a \in B \) or \( \forall b \in B \cdot P \overset{b}{\not\rightarrow} ) \)

\( P @ B \overset{a}{\rightarrow} P' @ B \)

Priority II. \( P \overset{a}{\rightarrow} P' \) and \( (a \not\in B \) or \( \forall b \not\in B \cdot P \overset{b}{\not\rightarrow} ) \)

\( P @ B \overset{a}{\rightarrow} P' @ B \)