Appendix A

FSP Quick Reference

A.1 Processes

A process is defined by a one or more local processes separated by commas. The definition is terminated by a full stop. `STOP` and `ERROR` are primitive local processes.

Example

```
Process = (a -> Local),
Local   = (b -> STOP).
```

Action Prefix **->**	If `x` is an action and `P` a process then `(x->P)` describes a process that initially engages in the action `x` and then behaves exactly as described by `P`.
Choice **|**	If `x` and `y` are actions then `(x->P|y->Q)` describes a process which initially engages in either of the actions `x` or `y`. After the first action has occurred, the subsequent behavior is described by `P` if the first action was `x` and `Q` if the first action was `y`.
Guarded Action **`when`**	The choice (**`when`** `B x -> P | y -> Q`) means that when the guard `B` is true then the actions `x` and y are both eligible to be chosen, otherwise if `B` is false then the action `x` cannot be chosen.
Alphabet Extension **+**	The alphabet of a process is the set of actions in which it can engage. `P + S` extends the alphabet of the process `P` with the actions in the set `S`.

Table A.1 – Process operators

A.2 Composite Processes

A composite process is the parallel composition of one or more processes. The definition of a composite process is preceded by **||**.

Example

```
||Composite = (P || Q).
```

Parallel	If `P` and `Q` are processes then `(P||Q)` represents the

Composition **||**	concurrent execution of `P` and `Q`.
Replicator **`forall`**	**`forall`** `[i:1..N] P(i)` is the parallel composition `(P(1) || … || P(N))`
Process Labeling **:**	`a:P` prefixes each label in the alphabet of `P` with `a`.
Process Sharing **::**	$\{a_1,..,a_x\}$`::P` replaces every label `n` in the alphabet of `P` with the labels $a_1.n,…,a_x.n$. Further, every transition `(n->Q)` in the definition of `P` is replaced with the transitions $(\{a_1.n,…,a_x.n\}\text{->}Q)$.
Priority High **<<**	`||C =(P||Q)<<`$\{a_1,…,a_n\}$ specifies a composition in which the actions $a_1,…,a_n$ have higher priority than any other action in the alphabet of `P||Q` including the silent action `tau`. In any choice in this system which has one or more of the actions $a_1,…,a_n$ labeling a transition, the transitions labeled with lower priority actions are discarded.
Priority Low **>>**	`||C=(P||Q)>>`$\{a_1,…,a_n\}$ specifies a composition in which the actions $a_1,…,a_n$ have lower priority than any other action in the alphabet of `P||Q` including the silent action `tau`. In any choice in this system which has one or more transitions not labeled by $a_1,…,a_n$, the transitions labeled by $a_1,…,a_n$ are discarded.

Table A.2 – Composite Process Operators

A.3 Common Operators

The operators in Table A.3 may be used in the definition of both processes and composite processes.

Conditional **`if then else`**	The process **`if`** `B` **`then`** `P` **`else`** `Q` behaves as the process `P` if the condition `B` is true otherwise it behaves as `Q`. If the **`else`** `Q` is omitted and `B` is false, then the process behaves as `STOP`.
Re-labeling **/**	Re-labeling is applied to a process to change the names of action labels. The general form of re-labeling is: /{*newlabel*_1/*oldlabel*_1,… *newlabel*_n/*oldlabel*_n}.
Hiding ****	When applied to a process `P`, the hiding operator \\ $\{a_1..a_x\}$ removes the action names $a_1..a_x$ from the alphabet of `P` and makes these concealed actions "silent". These silent actions are labeled `tau`. Silent actions in different processes are not shared.

Interface **@**	When applied to a process P, the interface operator @{a_1..a_x} hides all actions in the alphabet of P not labeled in the set a_1..a_x.

Table A.3 – Common Process Operators

A.4 Properties

Safety **property**	A safety **property** P defines a deterministic process that asserts that any trace including actions in the alphabet of P, is accepted by P.
Progress **progress**	**progress** P = {a_1,a_2..a_n} defines a progress property P which asserts that in an infinite execution of a target system, at least one of the actions a_1,a_2..a_n will be executed infinitely often.

Table A.4 – Safety and Progress Properties

A.5 *FLTL* – Fluent Linear Temporal Logic

Fluent **fluent**	**fluent** FL = <{s_1,…s_n}, {e_1..e_n}> **initially** *B* defines a fluent FL that is initially true if the expression *B* is true and initially false if the expression *B* is false. FL becomes true immediately any of the initiating actions {s_1,…s_n} occur and false immediately any of the terminating actions {e_1..e_n} occur. If the term **initially** *B* is omitted then FL is initially false.
Assertion **assert**	**assert** PF = FLTL_Expression defines an FLTL property.
&&	conjunction (*and*)
||	disjunction (*or*)
!	negation (*not*)
->	implication ((A->B) ° (!A || B))
<->	equivalence ((A**<->**B) ° (A->B)&&(B->A))
next time **X** ***F***	iff ***F*** holds in the next instant.
always **[]*F***	iff ***F*** holds now and always in the future.
eventually **<>*F***	iff ***F*** holds at some point in the future.
until ***P*** **U** ***Q***	iff ***Q*** holds at some point in the future and ***P*** holds until

	then.
weak until ***P* W *Q***	iff ***P*** holds indefinitely or ***P* U *Q***
`forall`	**`forall`** `[i:R] FL(i)` conjunction of FL(i)
`exists`	**`exists`** `[i:R] FL(i)` disjunction of FL(i)

Table A.5 – Fluent Linear Temporal Logic