Esterel Language Quick Reference Card

Revision 3.0

Conventions

`bold`	Esterel keyword	[]	Optional term
|	Alternative	::=	Expansion term
{}	Repeatable		Introduces one sequential control element

Introduces a sequential control element that can be easily removed by optimization

Equations *equation*::=

[**assert** | **next**] *sig_id* [**<=** *bool-exp*] [*if bool-exp*]	Pure or assert signal
[**next**] **?***sig_id* **<=** *exp* [*if bool-exp*]	Valued signal

Statements *stat*::=

{ *stat* }	Bracketed statement to unambiguously express control flow	
nothing	• Empty statement • Terminates instantaneously	
pause	• Unit delay – pause for one instant	
halt	• Never terminates • Can only be aborted	
open *{port-id,}* **in** *stat* **end** [**open**]	• Makes port id components visible in current scope	
loc **:=** *exp*	• Assigns the evaluation of an expression to a variable location, *loc* is for example X, Y[2], … • Assignment is instantaneous	
call *proc-id* ({*proc-arg*,})	• Instantaneous execution of an external procedure	
emit [**seq**] [**next**] [{] {*equation*,} [}]	• Terminates instantaneously • Concurrent or sequenced execution of equations • Equation blocks can be replicated using the *for … dopar* statement	
sustain [**seq**] [**next**] [{] {*equation*,} [}]	• Never terminates • Equations same as *emit* • Acts as a macro for: *loop* *emit {equation,} ; pause* *end loop*	
stat$_1$ **;** *stat*$_2$	• Sequence statement • *stat*$_2$ immediately starts as soon as *stat*$_1$ terminates	
loop *stat* **end** [**loop**]	• Infinite loop; immediately restarts its body when terminated • Never terminates (can be aborted or *exit*-ed) • *stat* must not be instantaneous (combinational)	
always *stat* **end** [**always**]	• Used to transform an instantaneous statement into a permanent one • *stat* should be instantaneous. Is equivalent to: *loop stat each tick*	
[**positive**] **repeat** [*count-id***:=**] *exp* **times** *stat* **end** [**repeat**]	• Finite loop; positive keyword asserts that the body will be executed at least once • *count-id* = *exp* down to 1. • *count-id* is read-only	
if *bool-exp* [**then** *stat*$_1$] [**else** *stat*$_2$] **end** [**if**]	• Branching according to the value of test expression • Test and branching are instantaneous	
if {**case** *bool-exp* **do** *stat*} [**default do** ***stat***] **end** [**if**]	Multiple branching tests; the first true signal expression in the list starts its statement	
stat$_1$ || *stat*$_2$	• Parallel statements • Lasts as long as one thread is still alive	
for *iter-id* **<** *static-exp* **dopar** *stat* **end** [**for**]	• Replication statement: generates *static-exp* copies of *stat* that run concurrently • *iter-id* iterator goes from *0* to *static-exp-1* and is read-only • *static-exp* must be statically evaluable	
for *iter-id* **in** [*static-exp1*.. *static-exp2*] **dopar** *stat* **end** [**for**]	• Same as above except that *iter-id* goes from *static-exp1* to *static-exp2*	
await *guard-exp*	• Terminates when guard expression is true • Instantaneous with *immediate* keyword • Macro for: *abort halt when guard-exp*	
await *guard-exp* **do** *stat* **end** [**await**]	• Starts *stat* when the guard expression is true • Is equivalent to: *await guard-exp; stat*	
await {**case** *guard-exp* **do** *stat*} **end** [**await**]	Multiple waiting; the first guard expression in the list that is true starts its statements	
[**weak**] **abort** *stat* **when** *guard-exp* [**do** *stat* **end** [**abort**]]	• Abortion; kills the statement when the guard expression is true • The weak variant starts the statement in the current instant when the guard expression is true, before killing it • Can be instantaneous with the *immediate* keyword	(1)
[**weak**] **abort** *stat* **when** { **case** *guard-exp* **do** {*stat* }} **end** [**abort**]	Multiple case abortion (strong or weak); the first guard expression in the list that is true starts its statement	(1)
[**weak**] **abort** *stat*$_1$ **after** *stat*$_2$ **end** [**abort**]	• *stat*$_1$ and *stat*$_2$ are concurrently started; *stat*$_1$ is aborted when *stat*$_2$ terminates • Immediately terminates if *stat*$_2$ immediately terminates	(1)
loop *stat* **each** *guard-exp*	• Temporal loop with body initially started. • *guard-exp* cannot be immediate • Is a macro for: *loop* *abort* *stat; halt* *when guard-exp* *end loop*	
every *guard-exp* **do** *stat* **end** [**every**]	• Temporal loop with initial waiting of true guard expression • *guard-exp* can be immediate • Macro for: *await guard-exp;* *loop* *stat* *each guard-exp*	
[**weak**] **suspend** *stat* **when** *bool-exp*	• Freezes the execution of a statement when guard expression is true. The weak variant keeps the combinational part running	
trap *except-list* **in** *stat* {**handle** *except-evt* **do** *stat*} **end** [**trap**]	• Mechanism to catch exceptions raised by the body statement	
exit *except-evt*	• Raises an exception	
finalize *stat*$_1$ **with** *stat*$_2$ **end** [**finalize**]	• Starts *stat*$_2$ as soon as *stat*$_1$ spontaneously terminates or is strongly or weakly aborted • *stat*$_2$ must be instantaneous	
signal {*sig-id* [{ **[exp]** }] [: [**temp** | **reg**[**1**]][[**value**] *type* [**combine**] [**init** *exp*]],} | {**port** *sig-id* **:** [**input**|**output**|**inputoutput**] *intf-unit-id*, } {**extends** [**input**|**output**|**inputoutput**] *intf-unit-id*,} [{*refinement-clause* ,}] **in** *stat* **end** [**signal**]	• Local declaration of a signal • Determines its scope • See *module-header* expansion for *refinement-clause*	
oracle {*sig-id*[**:** *type*],} **in** *stat* **end oracle**	• Local declaration of signal introducing model non-determinism • Determines its scope	
var {*var-id* : [**temp**] *type*[**:=** *exp*],} **in** *stat* **end** [**var**]	• Local declaration of a variable • Determines its scope • Cannot be shared between threads	
mcrun|**run** *inst-id*/*mod-unit* [[[[**weak**] **reset** *sig-id***;**] [**clock** *sig-id;*] *not for mcrun* [**constant** {*new/old*,}**;**] [**type** {*new/old*,}**;**] [**function** {*new/old*,}**;**] [**procedure** {*new/old*,}**;**] [[**signal**] {*new/old*,}**;**]*****]]	• Creates an instance of a module or a multi-clock unit with renaming of the interface elements, i.e., connects the handled current objects with the interface objects of the module • *new* can be an expression bracketed into parenthesis for input renaming. * new can be empty for signal renaming	

(1) Only weak abortion may introduce sequential element

Predefined Types

bool	Literals are: true = '1, false = '0
clock	Clock type
unsigned<N>	Denotes unsigned numbers from 0 to N-1
unsigned<[n]>	Is equivalent to *unsigned<2**n >*
unsigned	Is equivalent to *unsigned<[32] >*
unsigned<>	Only for constant definitions. unsigned<> = N is equivalent to unsigned<N+1> = N Literals can be written in dec, bin, oct or hex
signed<N>	Denotes signed numbers from: -2^{n-1} to $2^{n-1}-1$
signed<[n]>	Is equivalent to *signed<2**n >*
signed	Is equivalent to *signed<[32] >*
signed<>	Only for constant definitions. signed<> = N is equivalent to signed<N+1> = N
integer	Is equivalent to *signed* Signed literals are unsigned literals preceded by + or – for example: –123 or +0x10fa
float	Literals examples: 2.31f, .12f
double	Literals examples: 2.31, 1.2E-1
string	Literal example: "a \"string\""

Array Types

type **{[exp]}**	Examples: int[5] denotes an array of 5 integers, bool[4][6] denotes a 4*6 matrix of booleans
bool[n]	Denotes a bitvector

Expressions

guard-exp::= `[immediate` | *uns-exp* `times]` *bool-exp*

encoding-exp ::= `u2bin(`*uns-exp, bit-size*`)` |
`u2code<`*my-cod*`>(`*uns-exp*`,`*uns-size*`,`*bit-size*`)`
`u2gray(`*uns-exp, bit-size*`)` |
`u2onehot(`*uns-exp, bit-size*`)` |
`s2bin(`*sign-exp, bit-size*`)` |

decoding-exp ::= `bin2u(`*bit-vector, uns-size*`)` |
`bin2s(`*bit-vector, sign-size*`)` |
`code2u<`*my-cod*`>(`*bit-vect-exp*`,`*uns-size*`,`*bit-size*`)`
`gray2u(`*bit-vector, uns-size*`)` |
`onehot2u(`*bit-vector, uns-size*`)` |

predefined-sig ::= `tick` | `never`

Operators

`-`	Unary minus	Num
`-`	Subtraction	Num
`#`	Incompatibility	Boolean
`*`	Multiplication	Num
`**`	Power	Num
`/`	Division	Num
`[op]`	Array operator	Extension of operator to array level, examples: [+], [mux]
`+`	Unary plus	Num
`+`	Addition	Num
`<`	Less than	Num
`<=`	Less than or equal to	Num
`<=>`	Equivalence	Boolean
`<>`	Inequality	Any
`=`	Equality	Any
`=>`	Implication	Boolean
`>`	Greater than	Num
`>=`	Greater than or equal to	Num
`>> <<`	Right shift and left shift	Bitvector
`>>> <<<`	Signed right shift and signed left shift	Bitvector
`abs`	Absolute value	Signed
`and`	Conjunction	Boolean
`assert<N>`	Size assertion	Signed or unsigned
`assert_unsigned<N>`	Size and sign assertion	Signed or unsigned
`binsize(n)`	Number of bits to encode n in binary	Signed or unsigned
`extend`	Unsigned extension of bit vector to nbit	Bitvector
`mcat lcat`	msb and lsb concatenation	Bitvector
`mod`	Module	Unsigned
`mux`	Multiplexor	Any
`next`	Next status of registered signal	Signal status
`next(?`*sig-id*`)`	Next value of a registered valued signal	Signal value
`not`	Negation	Boolean
`or`	Disjunction	Boolean
`pre`	Pre status of standard signal, false at initial instant	Signal status
`pre(?`*sig-id*`)`	Previous value of a standard valued signal	Signal value
`pre1`	Pre status of standard signal, true at initial instant	Signal status
`reverse`	Reverses argument bits	Bitvector
`sat<N>`	Saturation	Signed or unsigned
`sextend`	Signed extension of bit vector to nbit	Bitvector
`trunc<[n]>`	Truncation	Signed or unsigned
`xor`	Exclusive or	Boolean

Data

Unit

`data` *data-unit-id* `:`
data-elem
`end data`
	Data unit definition

Declaration (data-elem ::=)

`[host` | `generic]` *type-id* | `type-id=enum{..}` | *`type;`*

`[host` | `generic]` *constant cst-id* : *type* [=exp];

`host` | `generic function` *fct-id* `(`[{*type*,}]`)` `:` *type;*

`host` | `generic procedure` *proc-id* `(`[{`in` | `out` | `inout` *type*,}]`);`

`extends [data]` *unit-id;*

`map` [*map-id* :] *type-id* {{*field-id* `[exp[..exp],}};`

Interface

Unit

`interface` *intf-unit-id* `:`
[*data-elem*]
*interface-elem*l
`end interface`
	Interface unit definition

Declaration (interface-elem::=)

`input` *sig-id* [{`[exp]`}] [`:[temp] [value]` *type*
`[init` *exp*`] [combine` *function*`] ];`

`output` *sig-id* [{`[exp]`}] [`:[temp` | `reg[1]] [value]` *type*
`[init` *exp*`] [combine` *function*`] ];`

`inputoutput` *sig-id* [{`[exp]`}] [`:[temp] [value]` *type*
`[init` *exp*`] [combine` *function*`] ];`

`port` *port-id*`:[mirror] [input|output|inputoutput]` *intf-unit-id* `;`

`extends [interface` | `data] [mirror] [input|output|inputoutput]` *unit-id;*

`input` | `output relation` [*rel-id*`:`] *combinational-sig-exp* `;`

Module

Unit

`[main] module` *mod-unit-id* : [*module-header*] *stat* `end module`	Module unit definition

Header (module-header::=)

[*data-elem*]
[*interface-elem*]
[{`refine` *sig-id* : [`temp`][`mem`] [`reg[1]`][`init` *exp*][`combine` *function*] `;`}]

Multi-clock Unit

`[main] multiclock` *mod-unit-id:*
[*module-header*] *(including clocks)*
`stat` *(reduced set of statements)*
`end multiclock`

Lexical Elements

id::= [\]*letter*{ [*underline*] *alphanumeric*}

unsigned-literal:= `0[b | o | x]` *hexint with optional '_' characters*

double-literal::=[*integer*] [. *integer*] [`E[+|-]`*integer*]

float-literal::=*double-literal* `F`

string-literal ::=*"a \"double quote\n"*

bool-literal ::= `'0 | '1`

bitvector-literal:= `'[b | x]` *hexint with optional '_' characters*

label ::= *operator* | *delayed_statement@id*|*string-literal*

comment ::= `//` *mono line comment* |
`/*` *multi line comments* `*/` |
`///` *mono line propagated comments* |
`/**` *multi line propagated comments* `*/`

ESTEREL LANGUAGE QUICK REFERENCE CARD

FOR
ESTEREL STUDIO 5.3

www.esterel-technologies.com

Last modified: 3-Nov-05