XLIA Documentation

Table of Contents

I. Introduction

This document presents the entry language of the symbolic execution platform Diversity[i]. This language, called **XLIA** (e**X**ecutable **L**anguage for **I**nteraction and **A**rchitecture), is designed for specifying the behavior of component-based systems.

In XLIA, components are referred to as machines. These machines are communicating, hierarchical, and heterogeneous, and their evaluation semantics can be customized to fit specific analysis or execution contexts.

Each component is defined through a set of sections, selected among the following:

- « param », where you can declare variables to be initialized when instantiating the machine,
- « returns », where you can declare variables containing return values, when dealing with procedures,
- « declaration », which can contain
    - user-type definitions:
        - structures, enumerations, arrays,
        - type aliases, etc.
    - typed variable declarations with their « modifier », constants, clocks, etc.
    - buffer declarations, for communication,
    - interaction point declarations, gathering “ports, signals and messages”,
- « machine », where submachines can be defined, in particular “statemachine” states,
- « transition », where can be defined transitions towards « submachines », when modelling transition systems,
- « moe », where can be implemented « machine » behaviours via primitives (init, run, enable, disable, etc…) that are usually automatically generated after compiling (e.g. with « statemachines »),
- « com » defines communication and interaction through route connections between machine ports.

Legend

In document, we take inspiration from the BNF to present the XLIA grammar.

Symbols for the grammar meta-language are in orange: brackets [ ] surround optional elements, + means that the preceding expression (possibly between parentheses) can be repeated one or several times, * that it can be repeated any number of times (including none). Terminal symbols - that is language keywords, are in red, and non-terminals in green. In black are variable names and standard non-terminals (e.g. statement, expression) that will not be recalled in this document.

II. Prologue

The XLIA language version used for the specification is given in this section.

@xlia< system , 1.0 > :

III. Machine

The machine is the basic component on which the XLIA language relies. It is defined by its sections, which will be described later in the document. A specification could be totally defined with machines but, for the sake of simplicity, the XLIA language provides several types of high-level machines:

- « system », the main machine representing the system,
- « statemachine », for a native (à la DSL) representation of symbolic state-transition machines,
- « procedure », for a simplified representation of standard procedures,
- « state », for a native and simplified representation of basic or composite statemachine states.

```
[ design ] [modifier] machine [ < [ moc: ] statekind > ] aMachineID {
    machine_section_list
}

design ::= model | prototype

modifier ::= input_enabled | timed

statekind ::= and | or

machine_section_list ::=
    [ @parameter: … ]
    [ @declaration: … ]
    [ @machine: … ]
    [ @transition: … ]
    [ @moe: … ]
    [ @com: … ]


instance machine [< [ model: ] aModelID > ] anInstanceID [ params ] ;

params ::= ( ([ paramID : ] expression ,)+ )
```

The « design » attribute

The XLIA language supports the « model-instance » paradigm: a machine can either be

- a « **model** », which can be instantiated, one or several times, such as in object-oriented programming,
- a model « **instance** », in which the model parameters can be fully or partially initialized, such as class builders in object-oriented programming,
- a « **prototype** », which is an alias for the declaration of a model and its first instance at the same time, and whose parameters can be initialized during its declaration.

By default, for the sake of simplicity, any machine will be a « prototype ».

The « modifier » attribute

Several kinds of machines can be defined thanks to the « modifier » attribute.

The « input_enabled » attribute

This attribute enables the definition of a communicating machine that can react to any incoming event, at any time. For example, an « input-enabled » statemachine must be able to react to any input in any of its states. The use of this attribute can have strong semantical consequences. For example, in the case of asynchronous communication via buffers, if a transition receives a message that is not expected at that moment, this message is simply lost.

The « timed » attribute

For a machine with the « **timed** » attribute, Diversity adds elements that enable handling of symbolic time.

Model of computation (moc)

For every composite machine, the « **moc**: » attribute can be used to define concurrency when evaluating its components (in the « @machine » section):

- « **and** » means that its components are concurrent with each other, such as orthogonal states in statecharts that is, at any time, all components are active ; by default, concurrency will be simulated by using « interleaving ».
- « **or** » means that, at any time, of all components at the same level, at least one is active.

By default, the model of computation of a machine is « **or** ». Atomic machines (i.e. non-composite) are not concerned.

Example 1

```
model machine < moc:or > myModelMachine {
@parameter:
	var int u;

@machine:
	machine myPrototypeSubmachine1 { ... }
	machine myPrototypeSubmachine2 { ... }
@moe:
	@run{
		if u == 42 {
			run myPrototypeSubmachine1;
		}
		else
			run myPrototypeSubmachine2;
		}
	}
...
}

instance machine < model:myModelMachine > myInstanceMachine( u:42 );
```

IV. System

In a XLIA specification, the main machine is called a « system ».

[modifier] system [< [moc:] statekind >] aSystemID {
machine_section_list
}

Attributes « modifier » and « moc » can still be used with a system.

Example 2

```
System< and > mySystem {
@machine:
	model machine< or > myModelMachine { ... }

	instance machine< myModelMachine > myInstanceMachine42( 42 );
	instance machine< myModelMachine > myInstanceMachine18( 18 );
}
```

Unlike in Example 1, here, labels such as “moc:” or “model:” are omitted.

V. Statemachine

A statemachine models symbolic state-transition machines or, more generally, symbolic transition systems.

```
[ design ] [modifier] statemachine [< [ moc: ] statekind> ] aStatemachineID {
    machine_section_list
}

instance statemachine [< [ model: ] aModelID > ] anInstanceID [ params ] ;
```

Model of computation (moc)

Statemachine « and »

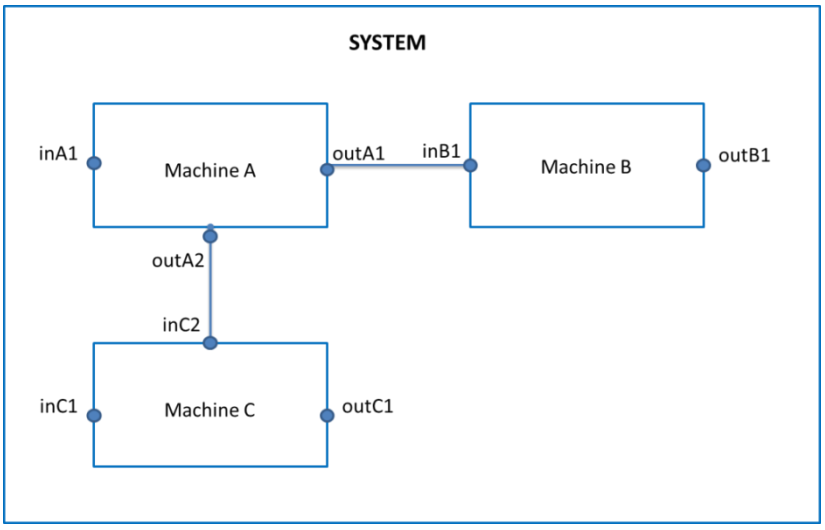
For concurrent state-transition machines whose scheduling (by default, operation « |i| ») can be defined with primitives « @irun».

The « @run{…} » primitive

It is the main primitive of XLIA machines, describing the evaluation model of machines in the system. It is called at each execution step of the system in a « top-down » manner: in the @run primitive, a machine calls (directly or not) the execution of its components via the instructions run or schedule.

In a model with state-transition machines, this primitive can be completely generated by Diversity's compiler according to semantical information given by the user (e.g. transition priorities).

Let us have a look at the following system:



The following XLIA code describes this system:

```
System@run{ |;;|
  run MachineA ;
  { |,|
    run MachineB ;
    run MachineC ;
  }
}
```

A system's behaviour is given by the evaluation of its @run primitive, which calls the sub-machine @run primitives. The scheduling operators are described further, in Section «

Block instructions ».

Such as in "dataflow" systems, input parameters « @param » can be given when calling a @run primitive, and output data « @returns » can be collected. Let's suppose that ports described above are associated with variables with the same name. Then, the above code can be refined into the following one:

```
System@run{ |;;|
  run  MachineA (inA1) --> (outA1, outA2) ;
  { |,|
    run  MachineB (outA1) --> (outB1) ;
    run  MachineC (inC1, outA2) --> (outC1)  ;
  }
}
```

```
@run [ ( ( input ,)+ ) ]  [ --> (  ( output ,)+ ) ]  block_statement
input  ::=  aTypeID  paramID  [ =  expression ]
          |  varID [ =  expression ]

output  ::=  aTypeID  paramID  [ =  expression ]
```

Input parameters of the @run primitive will be declared as « transient » variables of the machine (i.e. computation variables, instead of state variables). But any variable that has the primitive in its scope can be declared as an input parameter for the primitive, provided the varID is declared in the primitive type.

The « @schedule{…} » and « @concurrency{…} ».

NB. Initialization of such a machine ends with the synchronous initialization (operation « |and| ») of all its states.

Statemachine « or »

For sequential state-transition machines.

NB. Initialization of such a machine ends with the initialization of its following states:

- its only « initial » pseudo-state, if it exists ;
- its only « start » state, if it exists ;
- otherwise, all its basic and composite states (and not pseudo-states), in a non-deterministic way (operation « |/\| ») .

Example 3

```
statemachine < moc:or > myStatemachine {
@region:
	state< initial > myStateInitial {
		@transition:
			transition t0 --> mySimpleState { /*actions*/ }
	}
	state mySimpleState {
		@transition:
			transition t1 --> myFinalState { /*actions*/ }
	}
	state< final > myFinalState;

}
```

VI. State

A state is used to model of basic or composite states of composite statemachines. Here, section names can be omitted.

```
state [< [ moc: ] statekind > ] aStateID {
    machine_section_list | ( state_component )+
}

statekind ::= ( and | or ) | ( simple | start | final )
      | ( initial | terminal ) | ( junction | choice ) | ( fork | join ) | ( dhistory | shistory )

state_component ::=
     [ declaration_variable ]
     [ definition_state ]
     [ definition_transition ]
     [ impl_primitive_activity ]

impl_primitive_activity ::=
     [ @init{ … } ]
     [ @enable{ … } ]
     [ @disable{ … } ]
     [ @abort{ … } ]
     [ @irun{ … } ]
     [ @run{ … } ]
     [ @final{ … } ]
```

Model of computation (moc)

Basic states « simple | start | final |»

A basic state is the only kind of (non-composite) state in which a « statemachine<or> » can be after a computation step (i.e. evaluation or initialization step).

The « start » state is an alias for both the only « state<initial> » and the « state<simple> » in which the « statemachine<or> » will be after its initialization, i.e. after the evaluation of its initial transition.

A « final » state ends the evaluation of its container « statemachine<or> ». It has no outgoing transition. Its « @final{…} » primitive can be implemented (see Section**Erreur ! Source du renvoi introuvable.**).

Pseudo-states « initial | terminal | junction | choice | fork | join | dhistory | shistory »

They are standard pseudo-states used in state-transition machines.

NB. Only pseudo-states which kind « initial, terminal, junction, choice» are currently supported.

Body of a state

In a state, it is possible, in the following order:

- to declare local variables, constants, etc. (see Declarations: variables, constants, clocks, …);
- if it is a composite state, to define sub-states or outgoing transitions; note that a transition always belongs to its source state;
- to specify activities, i.e. actions to be done at key moments of its life cycle: initialization/finalization, activation/deactivation/abortion, evaluation, etc.

Activity primitives

In a symbolic transition system, activity primitives define actions that will be realised at key moments during the state's evaluation cycle. They are part of the whole set of primitives that define a machine's model of execution which can be implemented in section « @moe: ».

NB. Diversity analyses each specification and compiles it into "optimized pseudo code"» for its symbolic evaluation engine. It can sometimes complete the user code written in primitives. Unless specified, specification code will always be evaluated before Diversity generated code.

The « @init{…} » primitive

It can naturally – but not exclusively – be implemented for any composite, « initial » or « start » state. The user can define actions to be executed in the initialisation phase of the state that is, at the very beginning of the evaluation of the system. This is where the user can, for example, set an initial value to state variables. According to the kind of state, the initialisation will follow different patterns:

- « state< initial > » :
  - execution of its initialization primitive
  - execution of its « @run{…} » primitive, ending with the evaluation of its outgoing transitions
- « state< start > » :
  - execution of its initialization primitive

  N.B. The « @run {…} » primitive, and thus outgoing transitions, are not evaluated, unlike with initial states. So we stay in the start state after initialization. It is also the case with « simple » states.

- « statemachine< or > »:
  - execution of its initialization primitive
  - initialisation of its unique « state< initial > », or its unique « state< start > », or, if they do not exist, of its « state< simple > » or composite states in a non-deterministic way.
- « statemachine< and > » :
  - execution of its initialisation primitive

- simultaneous initialisation of all its states (i.e. synchronous parallelism); if the initialisation of one state fails, then the whole system's evaluation fails.

N.B. The user must make sure that all initialisations terminate. Otherwise, its symbolic transition system might get blocked and not evaluable later on.
For this reason, it is highly recommended to use mainly assignment statements in this phase;

- communication statements are not recommended,
- guarded statements are to be used with parsimony (the user must make sure that the logical condition is satisfiable).

The « @enable{…} » and @disable{…} » primitives

They correspond to Entry and Exit activities from UML Statemachines. They are evaluated, respectively, when activating and deactivating any state. The evaluation of a transition will follow this pattern: source state deactivation, action evaluation and then target state activation. Other scheduling possibilities will be available in further XLIA versions.

The « @abort{…} » primitive

It is triggered by abortion transitions (cf. The « @transition: », L'attribut : « abort » attribute) when deactivating the container state (instead of the @disable primitive triggered by other outgoing transitions). It is dedicated to composite states.

The « @irun{…} » and « @run{…} primitives

Usually, the user implements the « @irun{…} » code, whereas the « @run{…} » code is generated by Diversity's compiler.
The « @irun{…} » primitive corresponds to the Do activity of UML Statemachines: every attempt to evaluate a state starts with the evaluation of the « @irun{…} » if it is implemented, and goes on with the evaluation of the « @run{…} ». The effects of the « @irun{…} » evaluation are preserved even if the evaluation of the « @run{…} » fails. For instance, in the « @irun{…} » can be declared a clock counting the number of evaluation attempts of the state.
The « @run{…} » primitive evaluates outgoing transitions of its container state, according to the associated semantics. But the expert user can add code to be evaluated before outgoing transitions.

The « @final{…} » primitive

It is dedicated to:

- « statemachine< or > » with at least one « state< final > »; the activation of one of these « state< final > » will trigger the evaluation of the statemachine's « @final{…} » primitive;
- « statemachine < and > » whose sub-states, which are necessarily composite, all have at least one « state< final > »; when all the sub-states' « state< final > » are activated, then the statemachine's « @final{…} » primitive is evaluated;
- « state< final > » ; the state activation ends with the evaluation of the « @final{…} » primitive (just after the end of the « @enable{…} » primitive).

N.B. In composite states, the evaluation of « < final > » outgoing transitions is done as soon as their « @final{...} » primitive has been evaluated.

Example 4

```
// long version
state mySimpleState {
@transition:
       transition t1 --> myFinalState { /*actions*/ }
@moe:
       @enable{ /*entry-action, declenchée par les incoming transition*/ }
       @disable{ /*exit-action declenchée par les outgoing transition */ }
}

// short version
state mySimpleState {
       transition t1 --> myFinalState { /*actions*/ }

       @enable{ /*entry-action, declenchée par les incoming transition*/ }
       @disable{ /*exit-action declenchée par les outgoing transition */ }
}
```

In the following, we will use the short way for state definition, that is, without section names such as @transition or @moe.

VII. The body of a machine: sections

A machine is defined by its sections. The sections order is important for easiness of syntactical and semantical analysis:

- first, parameter declarations, state data (variables, buffers), and interaction points (ports, …);
- then, components (submachines);
- and finally, the model of execution and the model of communication.

The « @parameter: » section

When instantiated, a machine can receive parameters, that is data denoted by variables. These parameters must be declared in the « @parameter: » section.

Example 5

```
machine aMachineID{
@parameter:
       var int p1;
       var bool p2;
}
```

The « @returns: » section

A machine can also send data – denoted by variables, at the end of a computation step. These variables must be declared in the « @returns: » section.

Example 6

```
machine aMachineID{
@returns:
	var int v;
}
```

The « @declaration: » section

As usual, a machine can also be defined thanks to specific data, in addition to parameters. These data must be defined in the « @declaration: » section.

Each data in this section is associated with a visibility « modifier », which is « private » by default. For public data, the modifier must be explicitly set to « public ».

Example 7

```
machine aMachineID{
@declaration:
	var bool u;
	public var bool b;
}
```

The user can define several declaration sections, in which all declarations have the same visibility « modifier »: « private » or « public ». In that case, @declaration can be replaced by the name of the modifier.

Example 8

```
machine myMachine{
@private:
    const integer TMIN = 20;
    const integer TMAX = 25;
@public:
    var integer v1;
    var integer v2;
@public:
    port input p1;
    port output p2;
}
```

Type definition

It is the association of an alias (a type name) with an existing type.

```
@declaration:
        [modifier] type aNewTypeID aPrimitiveTypeDefinition ;

        [modifier] type aNewTypeID aTypeID ;

        [modifier] type aNewTypeID anArrayTypeDefinition ;
        [modifier] type aNewTypeID aCollectionTypeDefinition ;
        [modifier] type aNewTypeID aQueueTypeDefinition ;

        [modifier] type aNewTypeID anEnumerationTypeDefinition
        [modifier] type aNewTypeID aStructureTypeDefinition
        [modifier] type aNewTypeID aUnionTypeDefinition

modifier ::= public
```

Primitive types

They are standard predefined types, signed or not, which represent Booleans (**boolean** | **bool**), characters (**char**), strings (**string**), integers (**integer** | **int** | **uinteger** | **uint**), rationals (**rationnal** | **rat** | **urationnal** | **urat**), floats (**float** | **ufloat** | **double** | **udouble**), reals, etc.

By default, calculations on numbers are performed with arbitrary precision, according to « gnu/gmp ». But it is possible to give limits to types by defining an interval thanks to powers of two:

- [0 , 2^N] for unsigned types,
- [-2^{N-1} , $2^{N-1} - 1$] for signed types.

```
numeric_type < [ bit: ] integer<positive> >

numeric_type : integer<positive>
```

Thus,

- **int<bit:8>** is the set of all integer values between **-128** and **127**.
- **urat:9** is the set of all rational values between **0** and **512**.

Less symmetrical intervals can be defined via the « interval » type.

Interval types

Intervals are defined upon a « support », which must be a primitive numerical type. The standard ISO notation will be used to tell if the interval is open or closed, i.e. if its limits are included or not:

- left : open « **]** », closed « **[** »
- right : open « **[** », closed « **]** »

```
interval< aPrimitiveTypeID  isoIntervalDef >

isoIntervalDef [ < aPrimitiveTypeID > ]
```

In absence of any precision, the support will be deduced from the infimum and supremum type.

Array types

```
aTypeID [ integer<positive> ]
```

Collection types

A collection is a container for items of a given type. An optional maximal size can be given with a positive integer (« * » for infinite).

```
( array | vector | rvector | list | set | multiset ) < aTypeID [ , ( integer<positive> | * ) ] >
```

Example 9

```
var vector<UserEntry> userDataBase;
```

Queue types

Queues are ordered collections with specific addition and removal policies.

```
( fifo | lifo ) < aTypeID [ , ( integer<positive> | * ) ] >
```

Enumerated types

Enumerated types are sets of named values that usually behave as constants in the language.

```
enum {
        aSymbolID_1 [ = expression<numeric > ] ,
        …
        aSymbolID_N [ = expression<numeric > ]
}
```

Example 10

```
type PhoneOperation enum {CALL, TERMINATE_CALL, ACCEPT_CALL}
type CallReject enum {LINE_BUSY, UNKNOWN_NUMBER}
```

Structure types

They are standard *record* types.

```
struct {
        aVariableDeclaration_1 ;
        …
        aVariableDeclaration_N ;
}
```

Example 11

```
type PhoneCallAction struct {
     var int callerID;
     var PhoneOperation op;
}
```

Declarations: variables, constants, clocks, …

Every variable used in a machine must be previously declared, and it can be initialised with an expression. Several declarations sharing the same attributes can be grouped.

```
[modifier] var aTypeID aVariableID [ = initial_value ] ( ; | { rw_action } )

modifier ::= public | static | final | macro | volatile | unsafe

initial_value ::= ( = rvalue | ( rvalue ) )

rvalue ::= expression | { ( expression ,)+ }

rw_action ::= @on_write( aTypeID newVal ) block_statement
```

It is possible to combine several declarations:

```
[modifier] var { ( aTypeID aVariableID [ = initial_value ] ( ; | { rw_action } ) )+ }

[modifier] var< aTypeID > { ( aVariableID [ = initial_value ] ( ; | { rw_action } ) )+ }
```

For the sake of simplification, aliases can be used to replace some groups of « modifier var »:

- « **final var** » can be replaced by « **const** » when defining constants
- « **macro var** » can be replaced by « **macro** »
- « **clock var** » can be replaced by « **clock** »

The « modifiers » attributes of a variable are described in the following:

Constants: « final »

A constant is a variable which can be initialized but never modified. It can never be the left member of an assignment nor be used for data reception.

Macros: « macro »

A macro is a typed « variable-function » that can be used as any variable of the same type. When it is used, it must always have a value, which is a function of other variables, so that it can be valuated as a function. For example:

Example 12

```
var int { x ; y ; z ; }  // grouped declaration
macro int F = (x + y) ;
…
z = F ;  // literally equivalent to z = x + y ;
F ::= z – y ; // to syntactically reassign a macro
X = F ; // literally equivalent to z - y ;
```

The « volatile» attribute

This attribute indicates that the value of a public variable can be modified outside its normal visibility scope, for example by a parallel machine.

The « transient » attribute

This attribute indicates that a variable can only be used in computations, and never as a state variable of the machine. This information is used by heuristics such as those which try to anticipate redundant symbolic evaluations.

The « unsafe » attribute

This information is used by the XLIA compiler: if the variable is of a ranged type, the compiler will check all its uses and possibly generate a monitor to put constraints on expressions which are assigned to it.

Example 13

```
unsafe var  int:6  temperature;

is equivalent to

var  int:6  temperature; {
       @on_write( val ) {
              guard( (-32 < val) and (val < 31) );
       }
}
```

Buffers declarations

A buffer is a queue where messages are stored. A maximal size can be affected to each buffer.

```
[modifier] buffer buffer_def aBufferID ;

modifier ::= public |

buffer_def ::= ( fifo | lifo | set| multiset | ram ) [ < ( integer<positive> | * ) > ]
```

Combined declarations:

```
// buffers with different types
[modifier] buffer { ( buffer_def aBufferID ; )+ }

// buffers of the same type
[modifier] buffer< buffer_def > { ( aBufferID ; )+ }
```

Interaction points declaration

An interaction point – called « port » in XLIA, enables data transmission during communication operations. It can be a one-way point (input or output), or a two-way point (inout) for bidirectional communications. Every data type going through a port must be specified when declaring the port:

- they can be given a name,
- they can be associated with an expression to be sent by default, such as a macro, which is evaluated before transmission,
- they can be associated, by default, with the left part of an assignment, which will be evaluated during the reception operation, and used to store the data.

The « @bind » attribute binds these parameter-expressions to the port, and forbids any substitution in communication statements.

N.B. **Signals** must be declared in a machine containing the machines where it is used, and they are always **inout**.

```
[modifier] (port | signal | message) port_io aPortID [ params ] ;

modifier ::= public

port_io ::= input | output | inout

params ::= ( ( param ,)+ )

param ::= aTypeID [ paramID ]
        | aTypeID [ @bind ] [ paramID : ] expression<macro>
```

Combined declarations:

```
// ports with different signatures
[modifier] (port | signal | message) { ( port_io aPortID [ params ] ; )+ }

// ports with the same direction
[modifier] (port | signal | message) < port_io > { ( aPortID [ params ] ; )+ }

// ports of the same signature
[modifier] (port | signal | message) < port_io [ params ] > { ( aPortID ; )+ }
```

The « @transition: » section

Transitions describe operational behaviours in peculiar machines, called symbolic state-transition machines.

This section can be included in a statemachine in order to gather output transition definitions.

```
[modifier] statemachine [< [ moc: ] statekind > ] aStatemachineID {
…
@transition:
…
}
```

But for the sake of simplicity, in a « state », the name of the « @transition: » section can be avoided.

```
[modifier]  transition [< ( moc4trans ,)+ > ]   aTransitionID  moe4trans

modifier ::=  transient | …

moc4trans ::=  [ moc: ] transkind   |  [ prior: ] integer<positive>

transkind   ::=   ( simple | abort | final )   [ [  & ] else ]


moe4trans::=  block_statement  [ --> ]  ( aTargetStateID  ,)+  ;
                | -->  ( aTargetStateID  ,)+  (  block_statement  |  ;  )
```

The « modifier » attributes of a transition are described in the following.

The « transient » modifier

The specificity of a « transient » transition is that its evaluation is followed by the evaluation of its target state, whatever state it is, stable or not.

The model of computation

The « simple » attribute

By default, and as usual, a transition type is « simple ».

L'attribut : « abort »

A transition with the abort attribute is called an « abortion transition ». When the source state is composite, the abortion transition has priority on the state's internal transitions. Moreover, its evaluation will trigger the @abort primitive instead of the @disable primitive, as mentioned in Section State, ChapterThe « @abort{…} ».

The « final » attribute

It indicates finalisation transitions. Their evaluation starts when their source state, which is necessarily composite, has all its « state< final > » activated (see chapter on the **Erreur ! Source du renvoi introuvable.** primitive).

The « [&] else » attribute

It indicates the transition(s) with the lowest priority d'une classe donnée. It is used when one wants to give a deterministic evaluation of a state's output transitions, using logical constraints or explicit priorities (see following section).

The expression « transition< else > » is an alias for « transition< simple & else> ».

Priorities

They schedule transitions of the same kind (simple, abort, final, etc.) and coming from the same source state. Usually, the highest priority is 0, and the transition with the lowest priority will be assigned the highest number of all transitions. Two transitions with the same priority will be evaluated in a non-deterministic way.

Example 14

```
state mySimpleState {
       transition t0 --> targetState0 { /*actions*/ }
       transition< prior:1 > t1 --> targetState1 { /*actions*/ }
       transition< prior:2 > t2 --> targetState2 { /*actions*/ }

       transition< else >    t3 --> targetState3 { /*actions*/ }
}
```

Priority order is t1, then t2, then t3.

Example 15

```
state < moc:or > myCompositeState {
       state< initial > myStateInitial { /*transitions*/ }

       state mySimpleState { /* transitions */ }

       state< final > myFinalState{
           // simple transition
           transition t --> nalState { /*actions*/ }

           // final transition
           transition< final > t_final --> targetState1 { /*actions*/ }
       }
}
```

Transition t_final is triggered by the activation of "myFinalState".

The choice between transtition t and myCompositeState's (active state's) internal transitions is nondeterministic. But setting the attribute "abort" makes transition t prioritary:

Example 16

```
        state < moc:or > myCompositeState {
              state< initial > myStateInitial { /*transitions*/ }

              state mySimpleState { /* transitions */ }

              state< final > myFinalState;

              // Une transition
              transition<abort> t --> externalState1 { /*actions*/ }

              // une transition final
              transition< final > t_final --> externalState2 { /*actions*/ }
        }
```

The « @moe: » section

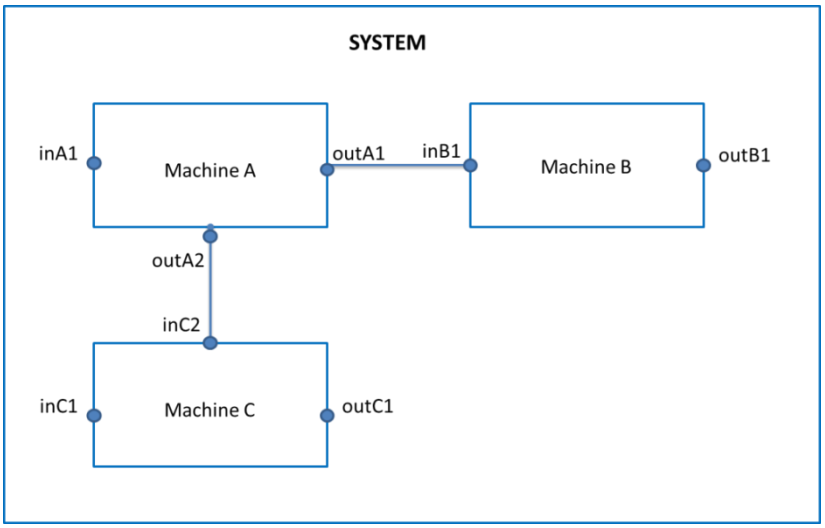
Most of this section's primitives are already described in the State chapter: « @init» and « @final», « @enable», « @disable», « @abort», « @irun».

The « @run{…} » primitive

It is the main primitive of XLIA machines, describing the evaluation model of machines in the system. It is called at each execution step of the system in a « top-down » manner: in the @run primitive, a machine calls (directly or not) the execution of its components via the instructions run or schedule.

In a model with state-transition machines, this primitive can be completely generated by Diversity's compiler according to semantical information given by the user (e.g. transition priorities).

Let us have a look at the following system:



The following XLIA code describes this system:

```
System@run{ |;;|
  run MachineA ;
  { |,|
     run MachineB ;
     run MachineC ;
  }
}
```

A system's behaviour is given by the evaluation of its @run primitive, which calls the sub-machine @run primitives. The scheduling operators are described further, in Section «
Block instructions ».

Such as in "dataflow" systems, input parameters « @param » can be given when calling a @run primitive, and output data « @returns » can be collected. Let's suppose that ports described above are associated with variables with the same name. Then, the above code can be refined into the following one:

```
System@run{ |;;|
  run  MachineA (inA1) --> (outA1, outA2) ;
  { |,|
     run  MachineB (outA1) --> (outB1) ;
     run  MachineC (inC1, outA2) --> (outC1)  ;
  }
}
```

```
@run [ ( ( input ,)+ ) ] [ --> ( ( output ,)+ ) ] block_statement
input  ::= aTypeID  paramID  [ = expression ]
         | varID [ = expression ]

output  ::= aTypeID  paramID  [ = expression ]
```

Input parameters of the @run primitive will be declared as « transient » variables of the machine (i.e. computation variables, instead of state variables). But any variable that has the primitive in its scope can be declared as an input parameter for the primitive, provided the varID is declared in the primitive type.

The « @schedule{…} » and « @concurrency{…} » primitives

These primitives are syntactic sugar to describe the scheduling of a machine's executable components:

- the « @schedule{…} » primitive will contain the evaluation politics for some (or all) of these components,

- the « @concurrency{…} » primitive will contain the scheduling operator for the remaining components (that are not in « @schedule{…} » nor in « @run{…} ») ; if this primitive is empty, the default operator is the interleaving operator « |i| ».

The link between policies described with these two primitives will be the weak sequence « |;;| ».

For example, suppose we have a machine M containing four executable components A, B, C and D, with primitives « @schedule{ |,| run A ; run C ; } » and « @concurrency{ |;;| } ».
Its implicit scheduling policy, triggered by primitive « @run{…} », will be

{ |;;| { |,| run A ; run C ; } { |;;| run B ; run D ; } }

which is equivalent, by transitivity of operator « |;;| », to

{ |;;| { |,| run A ; run C ; } run B ; run D ; }.

The « @com: » section

It describes interactions between machines, or between a machine and the environment, via ports and according to a given protocol.

```
@com[< protocol >]   :
   connect[< protocol >] { ( com_bus | com_port )+ }
   route[< protocol >] { (com_port )+ }

protocol ::=
  env  | rdv | ( multirdv [ , cast ] )
  buffer [ < type_buffer > | : bufferID ]

cast  ::=  unicast | multicast | broadcast

com_bus ::= ( input | output ) [< ( cast | type_buffer ) >]{ ( com_port )+ }
com_port ::= portID [ type_buffer ] ;
```

The communication protocol

The protocol tells whether the communication is between a machine and the environment (**env**), or between two machines (**rdv**). When several input/outputs are possible (**multirdv**), constraints can be added via the cast option:

- **unicast** implies that, at any time, there can only be one sender and one receiver,
- **multicast** implies that all outputs must be emissions, and all inputs must be receptions,

- **broadcast** implies that there are at least one sender and one receiver.

N.B. If there are several senders, messages should be without parameters: otherwise, it would not be possible to know which parameter is collected by a receiver.

The **buffer** keyword indicates that a message or signal must be stored in a buffer.

A default protocol can be given just after the section name.

Communication means

The **connect** keyword connects two ports or buses, possibly via a buffer. A com_bus a group of ports sharing some attributes, e.g. the cast option or a buffer.

Example 17

```
connect< env >{
        input Machine1->portA;
        output Machine2->portB;
        output Machine2->portC;
}

connect< rdv >{
        output Machine1->portD;
        input Machine2->portE;

}
```

In this example, the input port portA of Machine1 and the output port portB of Machine2 are connected to the environment. The output port portC of Machine1 is connected to the input port portD of Machine2, using the « rendez-vous » protocol.

The **route** keyword indicates where to store information in the case of a signal.

Example 18

```
machine aMachineID{
@declaration:
    buffer fifo b;
@com:
route <buffer:b1> {*}
```

In this example, all output data will be stored in buffer b, and all input data taken from b, unless if the target state (resp. the source state) is given in the output (resp. input) instruction: in that case, the route of the target (resp. the source) state has priority, if it exists.

N.B. The **rdv** mode is not implemented for signals, thus it is not compatible with keyword **route**.

VIII. Instructions

Foreword

In this section, we will describe the syntax of the main XLIA instructions and we will give an intuition of their symbolic evaluation semantics. Before that, we will introduce the notions of symbolic evaluation context and symbolic evaluation tree which represent all possible evaluation paths for an instruction.

The behaviour of a machine is described thanks to a set of instructions. An evaluation context **EC** (also called an evaluation environment) contains the information necessary for the symbolic evaluation of these instructions. It is composed of

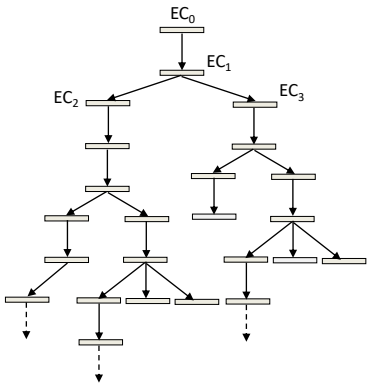
- a « Configuration State » **CS**, showing the system's state of control: for example, the set of active states for a state-transitions machine;
- a data part with
  - **VD** (« Variable Data »), the set of symbolic values associated with each variable,
  - **BD** (« Buffer Data »), the content of each communication buffer;
- a set of satisfiable constraints **PC** (« Path Condition »): it is a conjunction of logical conditions resulting from the symbolical evaluation of decision instructions , such as guarded or conditional instructions, that have guided the evaluation until configuration;
- information that enable a partial or global tracing of the evaluation: **ET** (« Evaluation Trace »);

A context **EC** is thus a tuple **(CS, VD, BD, PC, ET)**. In the following, we will sometimes omit some elements of the tuple when they are not useful for the evaluation of current instruction. For any instruction **stm**, notation « **EC -- stm --> {EC_1 EC_2 … EC_N}** » will mean that **stm**'s evaluation from context **EC** resulted in contexts **EC_1, …, EC_N**, representing all possible effects of **stm** on context **EC**. It will mean that the evaluation of **stm** from **EC** can lead whether to **EC_1**, or to **EC_2**, etc. The user can decide to put **stm** in the traces **ET_1**, …, **ET_N**. When the list of resulting contexts is empty, it means that the evaluation of the instruction has failed; otherwise, it terminates.

A symbolic evaluation tree is a tree built from an initial context **EC_0** and the evaluation of a "composite instruction", called process. This tree will be denoted by « (EC_0 {(EC_1 {(EC_a {…}) … (EC_k {…})}) … (EC_N{…})}) ». Intuitively, such a tree is built according to the following steps:

- Creation of an initial context EC_0 ;
- Evaluation of the machine's « @init{…} » primitive for initialisation;
- Repeated call of its « @run{…} » primitive, using, at each step, the context resulting from previous evaluation step.

The figure below is a graphical representation for the following symbolic evaluation tree: « (EC_0 {(EC_1 {(EC_2 {…}) (EC_3 {…})})}) ».



Block instructions

It defines a set of instructions ordered by an operator. By default, this operator is the standard sequence |;|. These block operators can be put into three categories: sequencing, scheduling and concurrency.

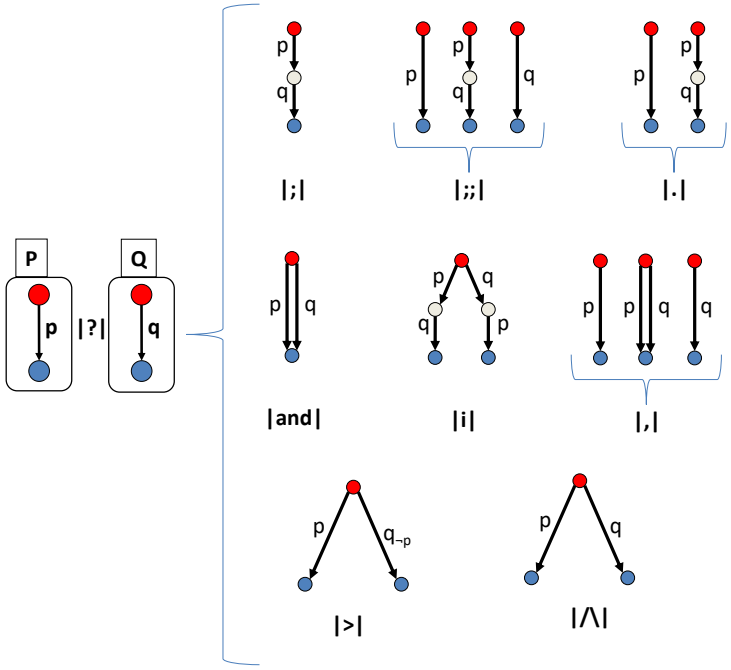
```
{ [ operator ] ( statement )+ }

operator ::=
        // sequencing
        |§|  |  |;|  |  |;;|  |  |.|

        // scheduling
        |>|  |  |xor|  |  |/\|

        // concurrencing
        |and|  |  |or|  |  |i|  |  |,|
```

Let **P** and **Q** be two processes. We say that a process terminates when it has been totally evaluated (without being interrupted by, for example, the failure of a guarded instruction), and that it fails otherwise.



There are as many evaluation possibilities as there are red dots (starting states). For instance, « **P |.| Q** » can be reduced to **P** if **Q** does not terminate, or to **P** followed by **Q**. If **P** does not terminate, **« P|.|Q »** does not terminate either.

Sequencing operators

The standard sequence, called strong sequence here: P |;| Q

Its evaluation terminates only if both **P** and **Q** terminate. Otherwise, no effects are preserved, and we go back to the initial context.

The weak sequence: P |;;| Q

This is not a standard sequence: its evaluation terminates if one of the two processes **P** or **Q** terminate. We try to evaluate **P**. If it terminates, then **Q** is evaluated from the resulting context, otherwise, from the initial context. This sequence can be used to create a partial order which avoids combinatorial explosion resulting from machine interleaving.

Sequence with side-effect: P |.| Q

This sequence terminates only if **P** terminates. If **Q** does not terminate, the resulting context will be **P**'s resulting context: **P** side-effects are preserved. For example, from context EC**<** CS, VD, BD, PC, ET >, if **Q** does not terminate, the evaluation of a transition labelled with « **P |.| Q** » will fail, but still, the result will be a new context EC'**<** CS, VD', BD', PC', ET' > with, at least, the same control CS as EC. The evaluation of this transition will be tried again from EC' at the next step.

Scheduling operators

Priority: P |>| Q

Process **P** is evaluated in the initial context. Process **Q** is evaluated in the initial context with in addition, if **P** terminates, the Boolean negation of conditions resulting from **P**'s evaluation. Thus, if both **P** and **Q** terminate, there are two possible resulting contexts

Indeterminism: P |/\| Q

Both processes **P** and **Q** are evaluated from the initial context. If both terminate, there will be at least two execution paths, showing the indeterminism.

Concurrency operators

Strong synchronous: P |and| Q

Both processes are evaluated from the initial context. If they both terminate, both contexts of resulting sets will be merged, providing that there are no concurrent accesses and the merge of their reachability constraints is satisfiable. Thus, we will simulate their strict simultaneous evaluation, i.e. their synchronous evaluation. Otherwise, the evaluation fails.

Interleaving: P |i| Q

Both processes are sequentially evaluated, considering every possibility: here **P** followed by **Q** and **Q** followed by **P**. Warning: this operator easily leads to an explosion of the number of execution paths.

Simple parallelism: P |,| Q

If both processes terminate, they are evaluated simultaneously if possible (for example, there must be no concurrent acces), otherwise the evaluation fails. If one of the processes does not terminate, the other one will be evaluated alone.

Assignment instructions

A « rvalue » expression is evaluated and assigned to the variable resulting of the evaluation of a « lvalue ». This variable is:

- the evaluation of the « lvalue », if it is itself an expression (for example, an access to an element in a collection),
- the « lvalue » itself if it is a simple variable – it cannot be a constant.

---

lvalue (= | :=) rvalue ;

Macro assignment

An expression « rvalue » is assigned to a « macro » without being evaluated.

---

lvalue<macro> ::= rvalue ;

Increment and decrement operations

[++ | --] lvalue ;

lvalue [++ | --] ;

Guarded instructions

The **guard** instruction evaluates a symbolic expression, checks whether it is satisfiable, and adds the result to Path Condition.

The **event** instruction evaluates an expression to **true** or **false** without using any solver. If it is not possible, the expression is evaluated to **false**.

---

guard expression< boolean >;

event expression< boolean >;

Example 19

```
guard patient_connected && (! Error) ;
guard timer != 0;
              }
event bad_connection;
event x>x_Max;
```

Communication instructions

---

input portID [((expression) +<,>)] [<-- machine] ;

output portID [((expression) +<,>)] [--> machine] ;

present portID

absent portID

Input and **output** instructions enable the sending and reception of messages or signals that might contain parameters. The sending machine – in the case of an input, or the receiving machine – in the case of an output, can be specified.

Keywords **present** and **absent** check whether a message is present or absent on a given port.

Example 20

```
@declaration{
   public port input p1;
   public port input p2(boolean);
   public port output p3;
   public port output p4(integer);
}
…
input  p1;
input  p2(x);
output p3 --> m2;
output p4(6,25);
```

Conditional instructions

They are standard conditional instructions.

if expression< boolean > { statement }

(elseif expression< boolean > { statement })*

[else { statement }]

Example 21

```
if cond1 {x = x+1;}
elseif cond2 {x = x+2;}
else {x = y;}
```

Iteration instructions

They are standard **for**, **while** and **do** instructions.

for

(statement < assign > ; expression< boolean > ; statement < assign >;

|

statement < assign > ; expression< boolean > ; statement < assign >;)

{ statement }

while expression< boolean> { statement }

do { statement } while expression< boolean > ;

Example 22

```
for x=1 ; x<x_Max ; x=x+1; {…}

while (x < x_Max) {x=x+1;}

do {x=x+1;} while (x < x_Max);
```

Control and loop instructions

They are standard control and loop instructions.

break ;

continue ;

[i] The symbolic execution platform Diversity is developed by CEA LIST. An open-source version is distributed through the Eclipse Formal Modeling Project (E-FMP): https://projects.eclipse.org/projects/modeling.efm