

The Core Language

Revisions :

13.3.2000	original document
16.9.2001	updated by Avi Magid
7.10.2001	updated by Avi Magid and Shahar Dag
18.8.2002	updated by Shahar Dag
18.11.2002	updated by Shahar Dag

Table of contents

TABLE OF CONTENTS	2
DESCRIPTION	3
Transition	3
Semantics:	3
Examples:	3
Example 1	3
Example 2	3
Example 3	4
Module	4
Semantics:	4
Examples:	5
Simple module	5
Asynchronous combination of modules	5
Synchronous combination of modules	6
Partially synchronized modules	7
Renaming	8
The Core Program	9
Examples:	9
CORE GRAMMAR	10
Tokens	10
The Grammar	12
Some explanations	18

Description

The Core language consists of a set of modules. A module is a set of transitions. A transition is defined as a transformation of a state machine into another state machine.

Transition

A transition contains four elements:

1. Name - An identifier of the transition.
2. Enable - A predicate, which defines a pre-condition.
3. Relation - A predicate, which defines a post-condition.
4. Assign - A different way to describe a post-condition by multiple-assignment.

Semantics:

A transition can only be “executed” when the state-machine satisfies the enable predicate. The transition changes the state-machine in any way so that the new state-machine satisfies the relation predicate.

The assignment statement defines that after the transition is over, several variables would hold specific expression values.

Examples:

Example 1

```
TRANS ex1:
  enable:      x = true;
  relation:    x'  $\vee$  y' = true;
```

Transition, named “ex1”, can be executed only when x equals true. If the transition is picked, then by the end of the transition the new values of x and y satisfy the equation $x \vee y = \text{true}$.

Example 2

```
TRANS ex2:
  enable:      t = 5;
  assign:     t' := 6 ;      q' := q * (t - 1);
```

Transition, named “ex2”, can be executed only when t equals 5. If the transition is picked, then after the transition is done the new value of t is 6 and the new value of q is $q * (t - 1)$.

Example 3

```
TRANS ex3:
    enable:    t = 5;
    assign:    t' := 6;
    relation:  t' = q;
```

Transition, named “ex3”, can be executed only if $t=5$ and if $q=6$. We can translate the assign section and insert it into the relation predicator and get the following equivalent transition:

```
TRANS ex4:
    enable:    t = 5;
    relation:  t' = 6  $\wedge$  t' = q;
```

If enable condition is true but there is no operation that can make the relation condition true, then transition “ex4” can never be picked.

Module

A module contains

1. Name - An identifier of this module.
2. Parameters - A group of variables which contain the values of actual variables during instantiation of the module.
3. Defines - Usually used to make the code more readable.
4. Local variables - A group of variables which are manipulated within the module.
5. {Transition}* - Set of Transitions.

Semantics:

A module defines a set of variables with or without initial values. The module defines a set of transitions which modify these variables in a non-deterministic fashion. The set of transitions can be defined in two ways:

1. A straightforward way: all transitions of the set are written word for word in the module.
2. A combination (not recursive) of several modules.

Examples:

Simple module

A module that contain only variables and transitions. The module may get external arguments to control it's bahaviure.

```
MODULE Bomb (Timer: integer)
{
    VAR
        Boom:          boolean INITVAL false;

    TRANS tick:
        enable:        Timer > 0;
        assign:        Timer' := Timer - 1;

    TRANS blowup:
        enable:        Timer = 0;
        assign:        Boom' := true;
}
```

Asynconus combination of modules

Asynconus combination is like taking all the variable and transitions from all the modules and put them together.

```
MODULE A ()
{
    VAR
        a:              integer INITVAL 6;

    (B (a) ||| C (a))
}
```

```
MODULE B (x: integer)
{
    TRANS hello:
        enable:        true;
        assign:        x' := 3;
}
```

```
MODULE C (x: integer)
{
    TRANS hello:
        enable:        true;
        assign:        x' := 7;
}
```

It would appear that there is a name conflict, but there isn't any. The "hello" in B and the "hello" in C, are known to be different transitions.

Synchronus combination of modules

When we synchronously combine modules we mean that whenever a transition from the first module is executed, a transition from the second module is executed. We do not mind which transition is selected from the two modules.

```
MODULE A ()
{
  VAR
    a:      integer INITVAL 0;
    b:      integer INITVAL 0;

    (First_M (a) || Second_M (b))
}
```

```
MODULE First_M (x: integer)
{
  TRANS hello:
    enable:    true;
    assign:    x' := 1;
}
```

```
MODULE Second_M (x: integer)
{
  TRANS shahar:
    enable:    true;
    assign:    x' := 1;

  TRANS dag:
    enable:    true;
    assign:    x' := 2;
}
```

In this example whenever hello (from the first module) is executed, either shahar or dag (from the second module) is also selected

Partially synchronized modules

Here only the defined pairs of transitions must be selected together, all other transitions are selected asynchronously.

```
MODULE A ()
{
  VAR
    a:          integer INITVAL 0;
    b:          integer INITVAL 0;

    (First_M (a) |(hello, shahar) (good_by, dag)| Second_M (b))
}

```

```
MODULE First_M (x: integer)
{
  TRANS hello:
    enable:    true;
    assign:    x' := 1;

  TRANS good_by:
    enable:    true;
    assign:    x' := 2;
}

```

```
MODULE Second_M (x: integer)
{
  TRANS shahar:
    enable:    true;
    assign:    x' := 1;

  TRANS dag:
    enable:    true;
    assign:    x' := 2;
}

```

In this example if hello (from the first module) is executed than shahar (from the second module) is also selected, but if good_by (from the first module) is executed than dag (from the second module) is selected.

Renaming

There is a conflict with “hello” in the following Core program:

```
MODULE A ()
{
  VAR
    a: integer INITVAL 6;

  ((B (a) ||| C(a)) |(hello, Yallow)| D())
}
```

```
MODULE B (x: integer)
{
  TRANS hello:
    enable:      true;
    assign: x' := 3;
}
```

```
MODULE C (x: integer)
{
  TRANS hello:
    enable: true;
    assign: x' := 7;
}
```

```
MODULE D ()
{
  TRANS Yallow:
    enable: true;
    assign: x' := 7;
}
```

The conflict is in module A: ((B (a) ||| C(a)) |(hello, Yallow)| D()). Which hello in the pair ((B (a) ||| C (a)) are we referring to?

For that, we need a mechanism of renaming: Let’s rename the “hello” transition in module B to “Bhello”, and “hello” transition in module C to “Chello”. The same thing would be done with “world”. The only change is in module A:

```
MODULE A ()
{
  VAR
    a: integer INITVAL 6;

  ((B (a) [hello -> Bhello, world -> Bworld] |||
  C (a) [hello -> Chello, world -> Cworld])
  |(Bhello, Yallow)|
  D())
}
```

The Core Program

A Core program consists of at least one module. The module from which everything starts (analogue to function main in C) is module SYSTEM. The Core program can define global variables, global defines, global constants and has the ability to define new types of variables.

Examples:

```
HOLD_PREVIOUS

VAR
    X : integer INITVAL 5;

MODULE SYSTEM ()
{
    ( M1(4) ||| M2(6) )
}

MODULE M1 (I: integer)
{
    TRANS a:
        enable: I < X;
        assign: X' := I;
}

MODULE M2 (I: integer)
{
    TRANS b:
        enable: I > X;
        assign: X' := I;
}
```

This Core program is equal to:

```
HOLD_PREVIOUS

VAR
    X : integer INITVAL 5;

MODULE SYSTEM ()
{
    TRANS a:
        enable: 4 < X;
        assign: X' := 4;

    TRANS b:
```

```

        enable: 6 > X;
        assign: X' := 6;
    }

```

Core Grammar

(The Core language is case sensitive).

Tokens

<u>Token name</u>	<u>Regular expression</u>	<u>Comments</u>
AND	"\ \\"	
ARRAY	"array" "ARRAY"	Type
ASSIGN	"assign"	Reserved word.
ASYN	" "	
ATOM	[&A-Za-z_][A-Za-z0-9_\\\$#-]*	Could be a name of a module, transition or a variable.
BOOL	"boolean" "BOOLEAN"	Type
COJOIN_FLAG	"COJOIN"	Reserved word.
COLON	":"	
COMMA	","	
CONST	"CONST"	Reserved word.
DEFINE	"DEFINE"	Reserved word.
DIVIDE	"/"	
ENABLE	"enable"	Reserved word.
ENUM	"ENUM"	Reserved word.
EQDEF	":="	
EQUAL	"="	
FALSEEXP	"false" "FALSE"	
GE	">="	
GT	">"	
HOLD_PREVIOUS	"HOLD_PREVIOUS"	Reserved word.
INITVAL	"INITVAL"	Reserved word.
INTEGER	"integer" "INTEGER"	Type
LB	"["	
LCB	"{"	
LE	"<="	
LINE	" "	
LP	"("	
LT	"<"	
MINUS	"_"	
MOD	"mod" "%"	
MODULE	"MODULE"	Reserved word.
MODULECOM	MODULE_COMMENT	Reserved word.
NEXT	">>"	

<u>Token name</u>	<u>Regular expression</u>	<u>Comments</u>
NOT	“!”	
NOTEQUAL	“!=”	
NUMBER	[0-9]+	
OF	“of” “OF”	Reserved word.
OR	“\V”	
PLUS	“+”	
RB	“]”	
RCB	“}”	
RELATION	“relation”	Reserved word.
RP	“)”	
SCALARSET	“SCALARSET”	Reserved word.
SEMI	[“,”[/t/n]*]*“;”	A separator is a series of “;” with or without spaces in between.
SIGN	“->”	
STRING	\ “[A-Za-z0-9]*\ \	A word which is bounded by “.”
SYNC	“ ”	
TIMES	“*”	
TRANS	“TRANS”	Reserved word.
TRUEEXP	“true” “TRUE”	
TWODOTS	“..”	
TYPE	“TYPE”	Reserved word.
VAR	“VAR”	Reserved word.

The Grammar

For readability the following tokens were replaced by their regular expressions:

TOKEN	replaced by
ASYNC	
COLON	:
COMMA	,
DIVIDE	/
EQDEF	:=
EQUAL	=
GE	>=
GT	>
LB	[
LCB	{
LE	<=
LINE	
LP	(
LT	<
MINUS	-
NOTEQUAL	!=
PLUS	+
RB]
RCB	}
RP)
SEMI	;
SIGN	->
SYNC	
TIMES	*
TWODOTS	..

The grammar:

- (1) begin ←
 global modules
- (2) global ←
 global HOLD_PREVIOUS
 | global CONST constlist
 | global TYPE typelist
 | global VAR varlist
 | global DEFINE deflist
 | ε

- (3) constlist ←
 | constlist ATOM : expr ;
 | ε
- (4) typelist ←
 | typelist typedef
 | ε
- (5) typedef ←
 | ATOM : number .. number ;
 | ATOM : SCALARSET (number) ;
 | ATOM : ENUM { Enumlist } ;
- (6) enumlist ←
 | enumlist , ATOM
 | enumlist ; ATOM
 | ATOM
- (7) varlist ←
 | varlist vardef
 | vardef
- (8) vardef ←
 | ATOM : typename initexpr ;
- (9) initexpr ←
 | INITVAL expr
 | ε
- (10) deflist ←
 | deflist ATOM := expr ;
 | ATOM := expr ;
 | ε
- (11) modules ←
 | module
 | modules module
- (12) module ←
 | MODULE ATOM (params) { modulebody } ending
- (13) modulebody ←
 | COJOIN_FLAG : expr ; modulebody
 | DEFINE defines modulebody
 | VAR declarations modulebody
 | MODULECOM STRING ending modulebody
 | transitions
 | modcombin

- (14) defines \leftarrow
 | defines ATOM := expr ;
 | ϵ
- (15) Params \leftarrow
 | ATOM : typename
 | params , ATOM : typename
 | params ; ATOM : typename
 | ϵ
- (16) declarations \leftarrow
 | declarations decl
 | ϵ
- (17) decl \leftarrow
 | ATOM : typename initexpr ;
- (18) typename \leftarrow
 | ARRAY [expr] OF typename
 | BOOL
 | INTEGER
 | ATOM
- (19) modcombin \leftarrow
 | (modcombin ||| modcombin)
 | (Modcombin || Modcombin)
 | (modcombin | pairs | modcombin)
 | ATOM modparams renaming
- (20) renaming \leftarrow
 | [Renamelist]
 | ϵ
- (21) renamelist \leftarrow
 | renamelist , rename
 | renamelist ; rename
 | rename
- (22) rename \leftarrow
 | pair -> ATOM
- (23) pairs \leftarrow
 | pairs , pair
 | pair

- (24) pair ←
 | (pair , pair)
 | (ATOM , pair)
 | (pair , ATOM)
 | (ATOM , ATOM)
 | ATOM
- (25) modparams ←
 | (modparamlist)
 | (
 | ε
- (26) modparamlist ←
 | modparamlist , atom
 | modparamlist ; atom
 | atom
- (27) transitions ←
 | transitions transition
 | transition
- (28) transition ←
 | TRANS ATOM : enable assign relation
 | TRANS ATOM : enable assign
 | TRANS ATOM : enable relation
- (29) enable ←
 | ENABLE : expr ;
- (30) assign ←
 | ASSIGN : assignlist ;
- (31) assignlist ←
 | assignlist , assignment
 | assignlist ; assignment
 | assignment
- (32) assignment ←
 | taggedatom := expr ;
- (33) relation ←
 | RELATION : expr ;

- (34) `expr` ←
 | constant
 | (Expr)
 | `expr + expr`
 | `expr - expr`
 | `expr / expr`
 | `expr * expr`
 | `expr MOD expr`
 | `expr = expr`
 | `expr != expr`
 | `expr <= expr`
 | `expr >= expr`
 | `expr < expr`
 | `expr > expr`
 | { neatomset }
 | `expr OR expr`
 | `expr AND expr`
 | NOT `expr`
- (35) `neatomset` ←
 | constant
 | neatomset , constant
 | neatomset ; Constant
- (36) `constant` ←
 | genconst
 | boolconst
- (37) `taggedatom` ←
 | atom NEXT
- (38) `atom` ←
 | ATOM
 | atom [`expr`]
- (39) `genconst` ←
 | atom
 | taggedatom
 | number
- (40) `boolconst` ←
 | FALSEEXP
 | TRUEEXP
- (41) `number` ←
 | NUMBER
 | + NUMBER
 | - NUMBER

(42) ending ←
| ;
ε

Some explanations

1	A core program consists of global data and modules.
2	<ul style="list-style-type: none"> • HOLD_PREVIOUS defines the functionality of an assignment. If it is present then for all variables not mentioned in the assignment retain their previous values. If it is not present then for all variables not mentioned in the assignment get a random value in their domain. • We can define constants, types, variables and definitions which are global.
3	Define global constants.
4	Define global types. This is the only place where one can define new types.
5	For now we can define 3 complex types: <ul style="list-style-type: none"> • Range: Values in an integer interval. • Scalar-Set: A group of integer values. • Enumeration.
6	Defines a enumeration list. Similar to C programming language.
7	Define global variables.
8	A definition of a variable consists of a name (identifier), a type and an initial value. If initial value is not present the variable receives a random value in the type domain.
9	Initial value can have an expression value. The value should be in the domain of the variable type.
10	A definition is analogous to an abbreviation of an expression.
11	The list of modules. One of the modules must be a module called SYSTEM. The SYSTEM module is the module from which the “tree of possibilities” starts.
12	A module consists of a name, parameters and a module body.
13	COJOIN_FLAG holds an expression, which must be true when a module is instantiated. A module has its own local definitions and variables. A module can consist of a set of transitions or a combination of modules.
14	A define consists of an identifier (a name) and an expression.
15	Formal parameters include the name and type.
16	Declaration of variables.
17	A variable declaration consists of an atom, the name of the type and an initial value (optional).
18	Core can define different types: <ul style="list-style-type: none"> • Boolean and Integer types which are the primitives. • Array of any type with a specific length (not necessarily a primitive).
19	Module combinations are: <ul style="list-style-type: none"> • A-synchronic (interleaving): A B. • Synchronic: A B. • Partial Synchronization: A (.....) B. <p>A module instantiation includes the module name, the actual parameters and renaming of transitions.</p>
20	A list of renames for transition names or pairs of transitions. Used to resolve name conflicts and for code simplicity.

21	A module instantiation can have several renames for its transitions.
22	A rename is an operation where we give a different name to a transition or to a pair. Used for abbreviation and also to differentiate two identical transitions in 2 different module instantiations of the same module.
23	A partial synchronization has to define the transitions, which have to be synchronic. A partial synchronization can define several pairs. Each pair defines a pair of transitions, which are synchronious.
24	A pair can be a transition name, a pair of transitions, or a pair of pairs.
25	A module instantiation can be with a list of actual parameters. If there are no formal parameters then the instantiation can hold an empty list of actual parameters “()”, or just the name of the module.
26	The actual parameters are simply expressions.
27	A module can be a set of transitions.
28	A transition has a name. A transition has an enable expression. A transition can have an assign or relation or both.
29	Enable is an expression that yields a boolean value.
30	As assign is constructed of a group of assignments. The group defines a multiple assignment. There is no order within the group.
31	An assignment list can have at least on assignment.
32	An assignment consists of a tagged variable (represents the variable value after the transition) and an expression.
33	Relation is an expression that yields a boolean value.
34	The structure of an expression.
35	An expression can be a set of constants.